COP 3530: Computer Science III
Summer 2005

Graphs – Part 3

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790

http://www.cs.ucf.edu/courses/cop3530/summer05

School of Computer Science
University of Central Florida

Ford’s Label Correcting Shortest Path Algorithm

- One of the first label-correcting algorithms was developed by Lester Ford. Ford’s algorithm is more powerful than Dijkstra’s in that it can handle graphs with negative weights (but it cannot handle graphs with negative weight cycles).

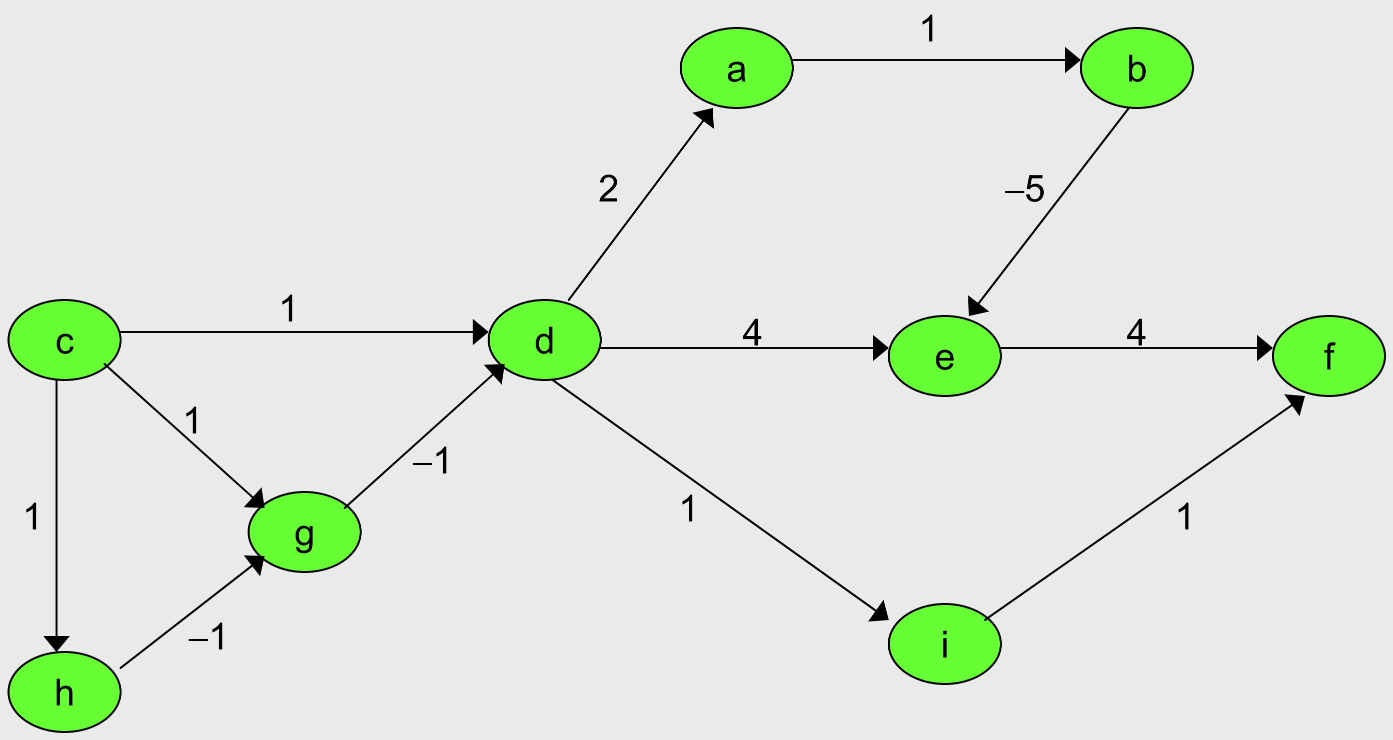
- To impose a certain ordering on monitoring the edges, an alphabetically ordered sequence of edges is commonly used so that the algorithm can repeatedly go through the entire sequence and adjust the current distance of any vertex if it is needed.

- The graph shown on slide 4 contains negatively weighted edges but no negative weight cycles.

Ford's Label Correcting Shortest Path Algorithm (cont.)

- As with Dijkstra's algorithm, Ford's shortest path algorithm also uses a table via dynamic programming to solve shortest path problems.
- We'll run through an example like we did with Dijkstra's algorithm so that you can get the feel for how this algorithm operates.
- We'll examine the table at each iteration of Ford's algorithm as the while loop updates the current distances (one iteration is one pass through the edge set).
- Note that a vertex can change its current distance during the same iteration. However, at the end, each vertex of the graph can be reached through the shortest path from the starting vertex.
- The example assumes that the initial vertex was vertex *c*.

Graph to Illustrate Ford's Shortest Path Algorithm



Graph for Ford's Shortest Path Algorithm Example

Ford’s Label Setting Algorithm

```
Ford (weighted simple digraph, vertex first)
    for all vertices v
        currDist(v) = $\infty$;
        currDist(first) = 0;
        while there is an edge (vu) such that
                [currDist(u) > currDist(v) + weight( edge(vu))]
                    currDist(u) = currDist(v) + weight(edge(vu));
```

Ford's Label Correcting Shortest Path Algorithm (cont.)

- Notice that Ford's algorithm does not specify the order in which the edges are checked. In the example, we will use the simple, but very brute force technique, of simply checking the adjacency list for every vertex during every iteration. This is not necessary and can be done much more efficiently, but clarity suffers and we are concerned about the technique at this point.
- Therefore, in the example the edges have been ordered alphabetically based upon the vertex letter. So, the edges are examined in the order of *ab, be, cd, cg, ch, da, de, di, ef, gd, hg, if*. Ford's algorithm proceeds in much the same way that Dijkstra's algorithm operates, however, termination occurs not when all vertices have been removed from a set but rather when no more changes (based upon the edge weights) can be made to any *currDist( )* value.
- The next several slides illustrate the operation of Ford's algorithm for the negatively weighted digraph on slide 4.

Initial Table for Ford's Algorithm

- Initially the *currDist(v)* for every vertex in the graph is set to ∞.

- Next the *currDist(start)* is set to 0, where *start* is the initial node for the path.
  - In this example *start* = vertex *c*.

- Edge ordering is: *ab, be, cd, cg, ch, da, de, di, ef, gd, hg, if*.

- The initial table is shown on the next slide.

Initial Table for Ford’s Shortest Path Algorithm

iteration →	initial	1				
vertices ↓						
a	∞					
b	∞					
c	0					
d	∞					
e	∞					
f	∞					
g	∞					
h	∞					
i	∞					

First Iteration of Ford’s Algorithm

- Since the edge set is ordered alphabetically and we are assuming that the start vertex is *c*, then the first iteration of the while loop in the algorithm will ignore the first two edges (*ab*) and (*be*).
- The first past will set the *currDist( )* value for all single edge paths (at least), the second pass will set all the values for two-edge paths, and so on.
- In this example graph the longest path is of length four so only four iterations will be required to determine the shortest path from vertex *c* to all other vertices.
- The table on slide 11 shows the status after the first iteration completes. Notice that the path from *c* to *d* is reset (as are the paths from *c* to *f* and *c* to *g*) since a path of two edges has less weight than the first path of one edge. This is illustrated in the un-numbered (un-labeled) column.

First Iteration of Ford's Algorithm (cont.)

- With the start vertex set as *C*, the first iteration sets the following:
  - *edge(ab)* sets nothing
  - *edge(be)* sets nothing
  - *edge(cd)* sets *currDist(d) = 1*
  - *edge(cg)* sets *currDist(g) = 1*
  - *edge(ch)* sets *currDist(h) = 1*
  - *edge(da)* sets *currDist(a) = 3* since *currDist(d) + weight(edge(da)) = 1+ 2 = 3*
  - *edge(de)* sets *currDist(e) = 5* since *currDist(d) + weight(edge(de)) = 1+ 4 = 5*
  - *edge(di)* sets *currDist(i) = 2* since *currDist(d) + weight(edge(di)) = 1+ 1 = 2*
  - *edge(ef)* sets *currDist(f) = 9* since *currDist(e) + weight(edge(ef)) = 5+ 4 = 9*
  - *edge(gd)* resets *currDist(d) = 0* since *currDist(d)+ weight(edge(gd)) = 1+ (-1) = 0*
  - *edge(hg)* resets *currDist(g) = 0* since *currDist(g)+ weight(edge(hg)) = 1+ (-1) = 0*
  - *edge(if)* resets *currDist(f) = 3* since *currDist(i) + weight(edge(if)) = 2+ 1 = 3*

Table After First Iteration

iteration →	initial	1				
vertices ↓						
A	∞	3	3			
B	∞	∞	∞			
C	0					
D	∞	1	0			
E	∞	5	5			
F	∞	9	3			
G	∞	1	0			
H	∞	1				
I	∞	2	2			

currDist(d) is initially set at 1 since edge (cd) is considered first.

Subsequently, when considering edge (gd) the currDist(d) can be reduced due to a negative weight edge and currDist(d) becomes 0.

First Iteration of Ford's Algorithm (cont.)

- Notice that after the first iteration the distance from vertex *c* to every other vertex, except *b* has been determined.
  - This is because of the order in which we ordered the edges. This means that the second pass will possibly set the distance to vertex *b* but the distance to all other vertices can only be reset if a new path with less weight is encountered.

Second Iteration of Ford's Algorithm

- The second iteration (second pass through edge set) sets the following:
  - *edge(ab)* sets *currDist(b)= 4* since *currDist(a) + weight(edge(ab)) = 3+ 1 = 4*
  - *edge(be)* resets *currDist(e)= -1* since *currDist(b)+weight(edge(be)) = 4 +(-5) = -1*
  - *edge(cd)* no change *currDist(d) = 0*
  - *edge(cg)* no change *currDist(g)= 0*
  - *edge(ch)* no change *currDist(h) = 1*
  - *edge(da)* resets *currDist(a) = 2* since *currDist(d) + weight(edge(da)) = 0+ 2 = 2*
  - *edge(de)* no change *currDist(e)= -1*
  - *edge(di)* resets *currDist(i) = 1* since *currDist(d) + weight(edge(di)) = 0 + 1 = 1*
  - *edge(ef)* no change *currDist(f) = 3*
  - *edge(gd)* resets *currDist(d)= -1* since *currDist(d)+ weight(edge(gd))= 0+ (-1) = -1*
  - *edge(hg)* no change *currDist(g) = 0*
  - *edge(if)* resets *currDist(f) = 2* since *currDist(i) + weight(edge(if)) = 1+ 1 = 2*

Table After 2nd Iteration

iteration →	initial	1		2		
vertices ↓						
A	∞	3	3	2		
B	∞	∞	∞	4		
C	0					
D	∞	1	0	–1		
E	∞	5	5	–1		
F	∞	9	3	2		
G	∞	1	0			
H	∞	1				
I	∞	2	2	1		

Third Iteration of Ford's Algorithm

- The third iteration makes the following updates to the table:
  - *edge(ab)* resets *currDist(b)= 3* since *currDist(a) + weight(edge(ab)) = 2+ 1 = 3*
  - *edge(be)* resets *currDist(e)= -2* since *currDist(b)+weight(edge(be)) = 3 +(-5) = -2*
  - *edge(cd)* no change *currDist(d) = -1*
  - *edge(cg)* no change *currDist(g)= 0*
  - *edge(ch)* no change *currDist(h) = 1*
  - *edge(da)* resets *currDist(a) = 1* since *currDist(d) + weight(edge(da))= (-1)+ 2 = 1*
  - *edge(de)* no change *currDist(e)= -2*
  - *edge(di)* resets *currDist(i) = 0* since *currDist(d) + weight(edge(di)) = -1 + 1 = 0*
  - *edge(ef)* resets *currDist(f) = 2* since *currDist(e) + weight(edge(ef)) = -2 + 4 = 2*
  - *edge(gd)* no change *currDist(d)= -1*
  - *edge(hg)* no change *currDist(g) = 0*
  - *edge(if)* resets *currDist(f) = 1* since *currDist(i) + weight(edge(if)) = 0+ 1 = 1*

Table After 3rd Iteration

iteration →	initial	1		2	3	
vertices ↓						
A	∞	3	3	2	1	
B	∞	∞	∞	4	3	
C	0					
D	∞	1	0	–1		
E	∞	5	5	–1	–2	
F	∞	9	3	2	1	
G	∞	1	0			
H	∞	1				
I	∞	2	2	1	0	

Fourth Iteration of Ford's Algorithm

- The fourth iteration makes the following updates to the table:
  - *edge(ab)* resets *currDist(b)= 2* since *currDist(a)* + *weight(edge(ab)) = 1+ 1 = 2*
  - *edge(be)* resets *currDist(e)= -3* since *currDist(b)+weight(edge(be)) = 2 +(-5) = -3*
  - *edge(cd)* no change *currDist(d) = -1*
  - *edge(cg)* no change *currDist(g)= 0*
  - *edge(ch)* no change *currDist(h) = 1*
  - *edge(da)* no change *currDist(a) = 1*
  - *edge(de)* no change *currDist(e)= -3*
  - *edge(di)* no change *currDist(i) = 0*
  - *edge(ef)* no change *currDist(f) = 1*
  - *edge(gd)* no change *currDist(d)= -1*
  - *edge(hg)* no change *currDist(g) = 0*
  - *edge(if)* no change *currDist(f) = 1*

Table After 4th Iteration

iteration →	initial	1		2	3	4
vertices ↓						
A	∞	3	3	2	1	
B	∞	∞	∞	4	3	2
C	0					
D	∞	1	0	–1		
E	∞	5	5	–1	–2	–3
F	∞	9	3	2	1	
G	∞	1	0			
H	∞	1				
I	∞	2	2	1	0	

Fourth Iteration of Ford’s Algorithm

- A fifth and final iteration is needed (its not shown in the table) which upon ending will terminate the algorithm as no changes will be made to the table on the fifth iteration. Since the fourth iteration reset only the *currDist( )* for vertices *b* and *e*, the only possible changes that could be made to the table during the fifth iteration would be to those same vertices again since these two did not affect the distance to any other vertex during the previous iteration. The fifth and final iteration is shown below:

- *edge(ab)* no change *currDist(b)= 2*
- *edge(cd)* no change *currDist(d) = -1*
- *edge(ch)* no change *currDist(h) = 1*
- *edge(de)* no change *currDist(e)= -3*
- *edge(ef)* no change *currDist(f) = 1*
- *edge(hg)* no change *currDist(g) = 0*

edge(be) no change *currDist(e)= -3*

edge(cg) no change *currDist(g)= 0*

edge(da) no change *currDist(a) = 1*

edge(di) no change *currDist(i) = 0*

edge(gd) no change *currDist(d)= -1*

edge(if) no change *currDist(f) = 1*

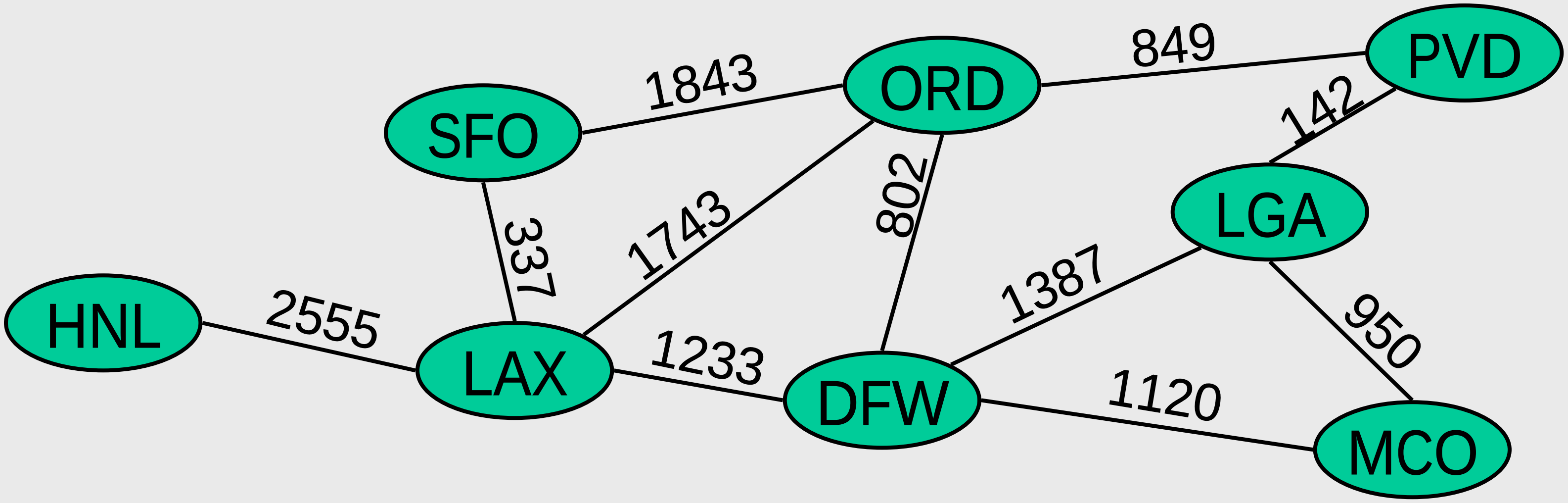
Comments on Ford’s Shortest Path Algorithm

- As you can see having stepped through the execution of Ford’s algorithm, the run-time is dependent on the size of the edge set.

- Ford’s algorithm works best if the graph is sparse and less well if the graph is relatively dense.

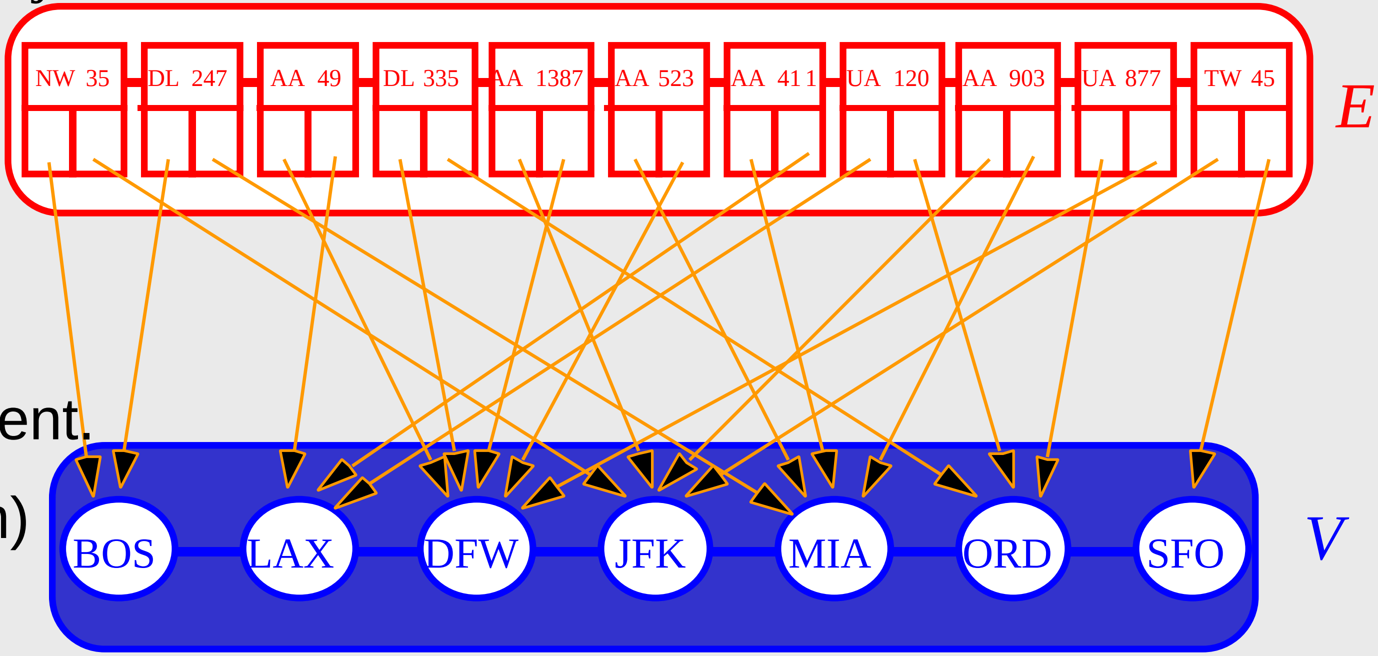
Graph: Example

- A vertex represents an airport and stores the three-letter airport code
- An edge represents a flight route between two airports and stores the mileage of the route



Edge List

- The edge list structure simply stores the vertices and the edges into two containers (ex: lists, vectors etc..)
- each edge object has references to the vertices it connects.



Easy to implement.

Space = O(n+m)

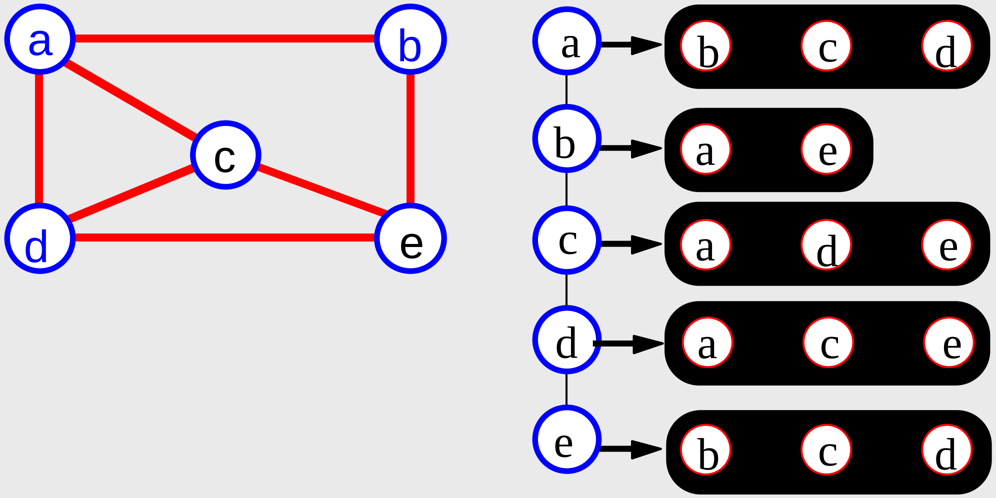
Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence. Time: O(m)

Adjacency List (traditional)

- adjacency list of a vertex v:

  sequence of vertices adjacent to v

- represent the graph by the adjacency lists of all the vertices



a → b c d

b → a e

c → a d e

d → a c e

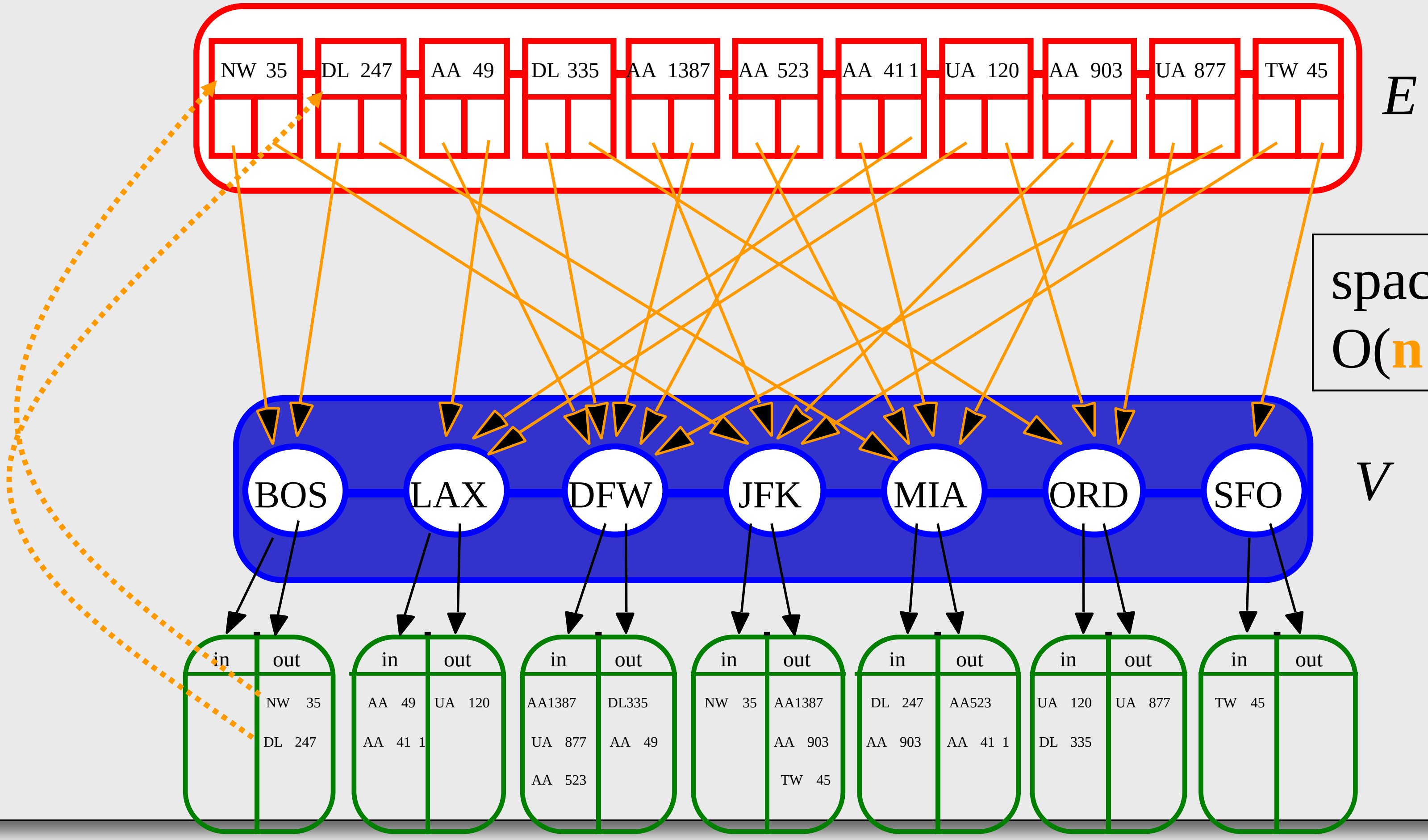
e → b c d

Space =

$\Theta(\mathbf{n} + \Sigma\, \mathbf{deg(v)}) = \Theta(\mathbf{n} + \mathbf{m})$

Adjacency List (modern)

- The adjacency list structure extends the edge list structure by adding incidence containers to each vertex.

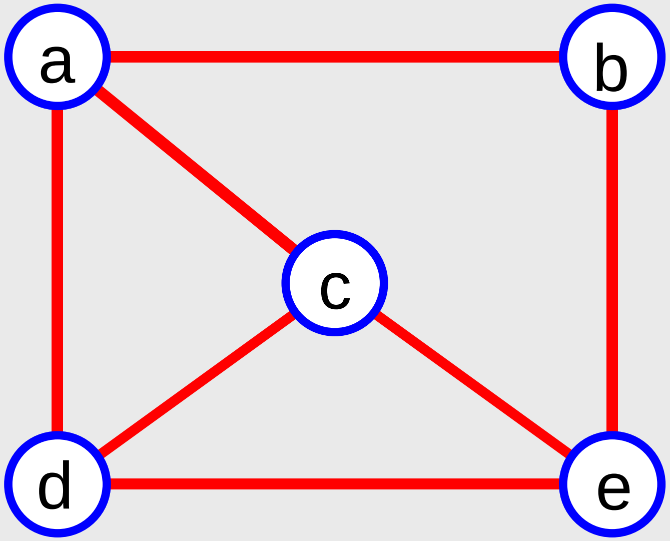


space is O(**n** + **m**).

Performance of the Adjacency List Structure

size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	O(1)
incidentEdges(v), inIncidentEdges(v), outIncidentEdges(v), adjacentVertices(v), inAdjacentVertices(v), outAdjacentVertices(v)	O(deg(v))
areAdjacent(u, v)	O(min(deg(u), deg(v)))
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection,	O(1)
removeVertex(v)	O(deg(v))

Adjacency Matrix (traditional)



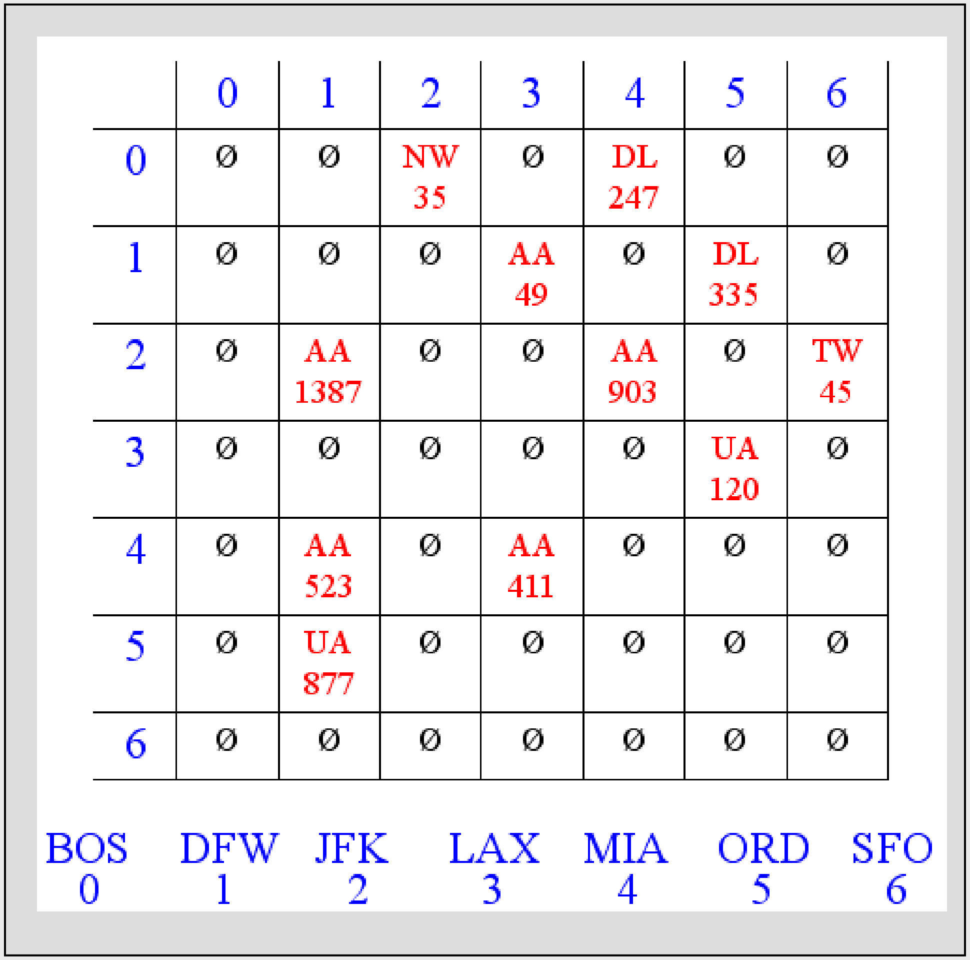
	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

- matrix M with entries for all pairs of vertices
- M[i,j] = true means that there is an edge (i,j) in the graph.
- M[i,j] = false means that there is no edge (i,j) in the graph.
- There is an entry for every possible edge, therefore:

$$\text{Space} = \Theta(n^2)$$

Adjacency Matrix (modern)

- The adjacency matrix structures augments the edge list structure with a matrix where each row and column corresponds to a vertex.



	0	1	2	3	4	5	6
0	Ø	Ø	NW 35	Ø	DL 247	Ø	Ø
1	Ø	Ø	Ø	AA 49	Ø	DL 335	Ø
2	Ø	AA 1387	Ø	Ø	AA 903	Ø	TW 45
3	Ø	Ø	Ø	Ø	Ø	UA 120	Ø
4	Ø	AA 523	Ø	AA 411	Ø	Ø	Ø
5	Ø	UA 877	Ø	Ø	Ø	Ø	Ø
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø

BOS	DFW	JFK	LAX	MIA	ORD	SFO
0	1	2	3	4	5	6

Graph Traversal

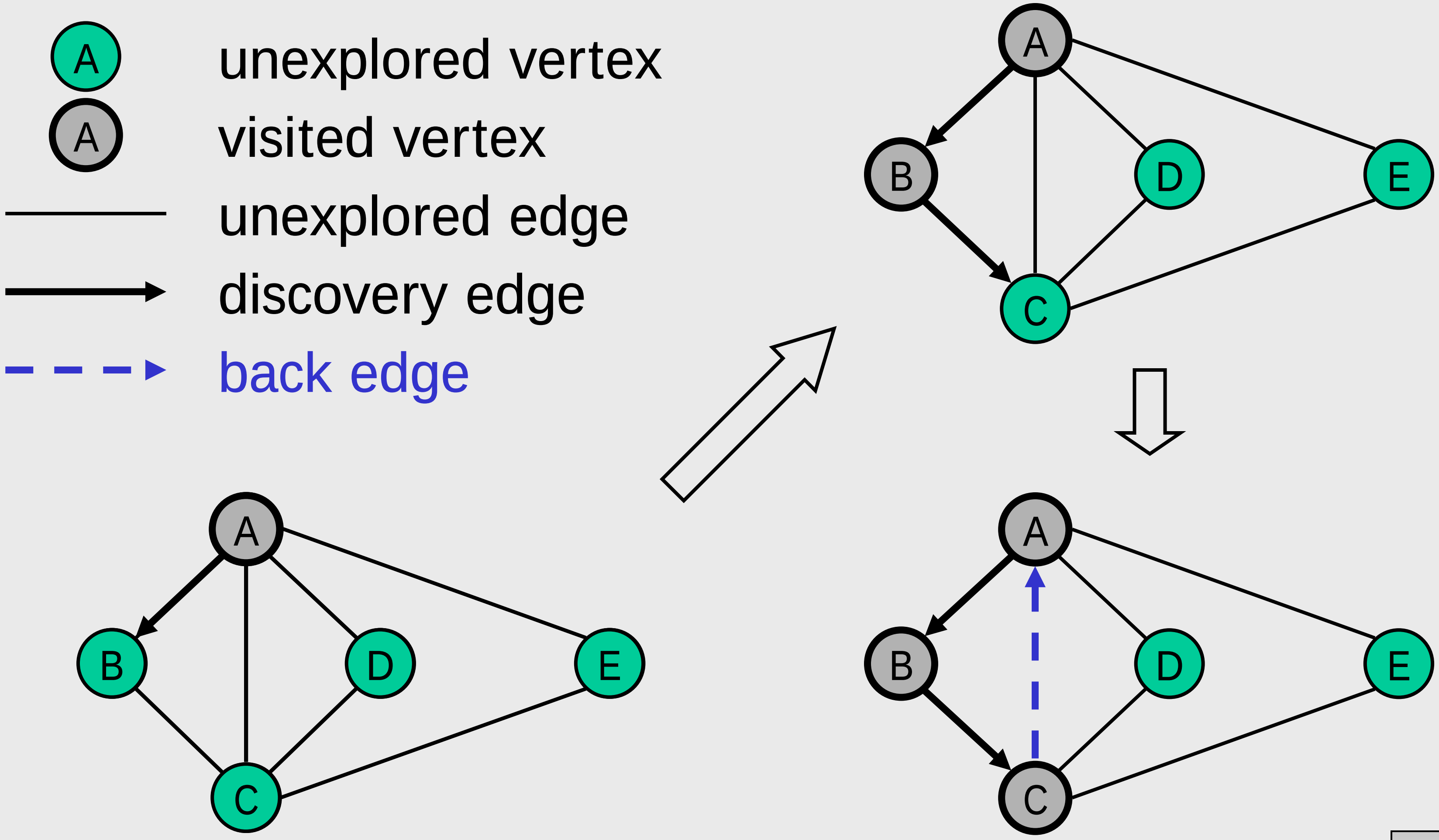
- A procedure for exploring a graph by examining all of its vertices and edges.

- Two different techniques:
  - Depth First traversal (DFT)
  - Breadth First Traversal (BFT)

Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G

Example



🟢 unexplored vertex

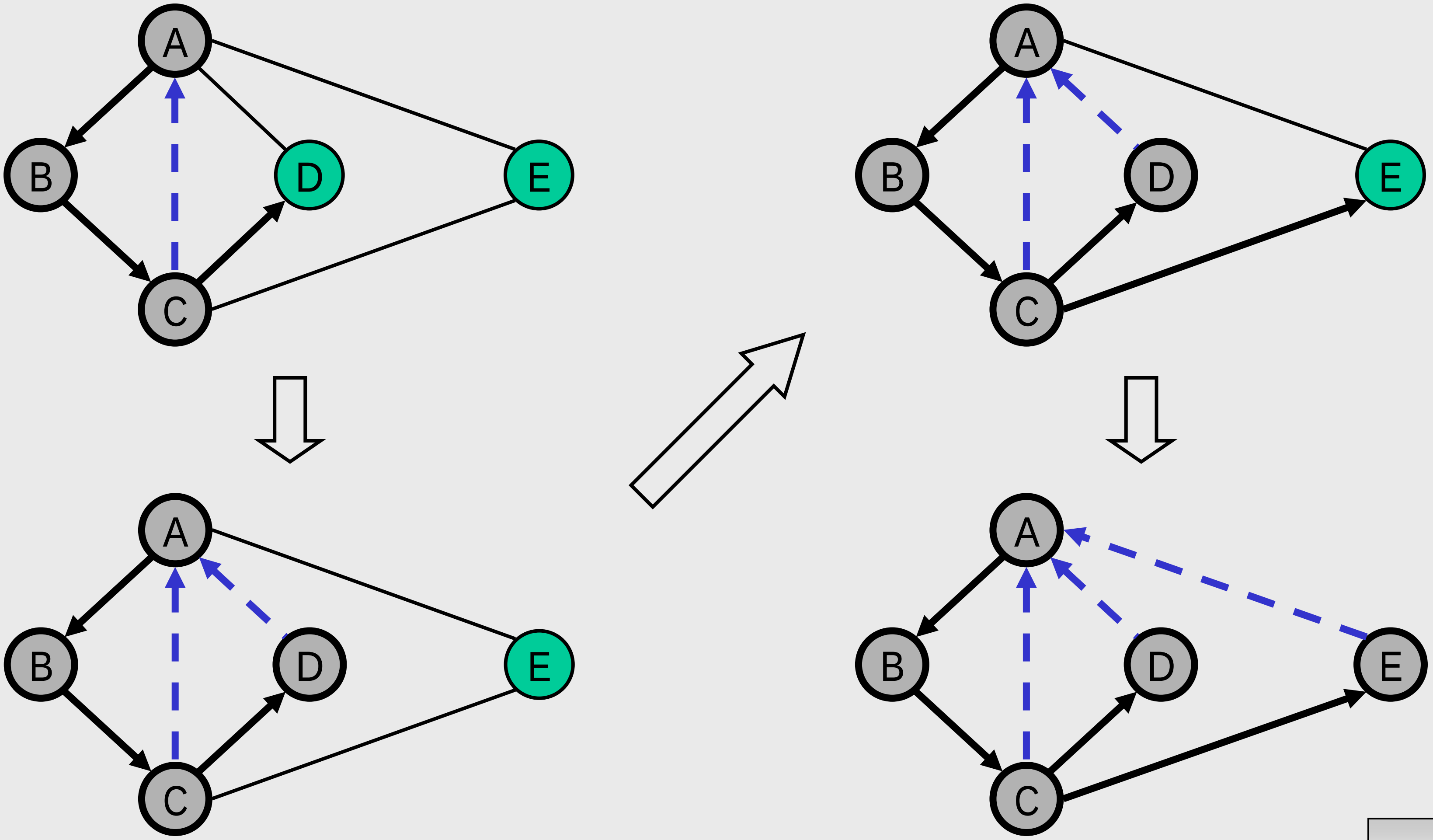
⚪ visited vertex

── unexplored edge

⟶ discovery edge

- - ▸ back edge

Example (cont.)



DFS Algorithm

- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

```
Algorithm DFS(G)
  Input graph G
  Output labeling of the edges of G
    as discovery edges and back edges
  for all  u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all  e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all  v ∈ G.vertices()
    if  getLabel(v) = UNEXPLORED
      DFS(G, v)
```

```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G  in
      the connected component of v as
      discovery edges and back edges
  setLabel(v, VISITED)
  for all  e ∈ G.incidentEdges(v)
    if  getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

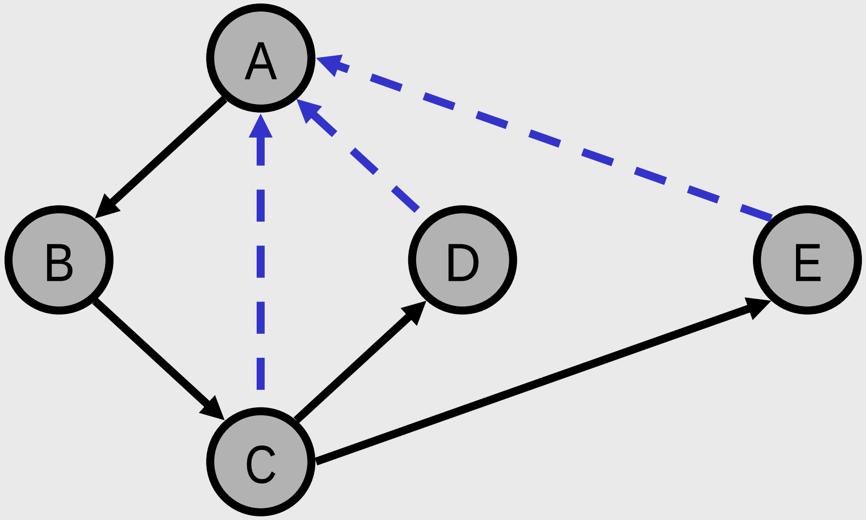
Properties of DFS

Property 1

DFS(***G*****, *v***) visits all the vertices and edges in the connected component of ***v***

Property 2

The discovery edges labeled by ***DFS***(***G*****, *v***) form a spanning tree of the connected component of ***v***



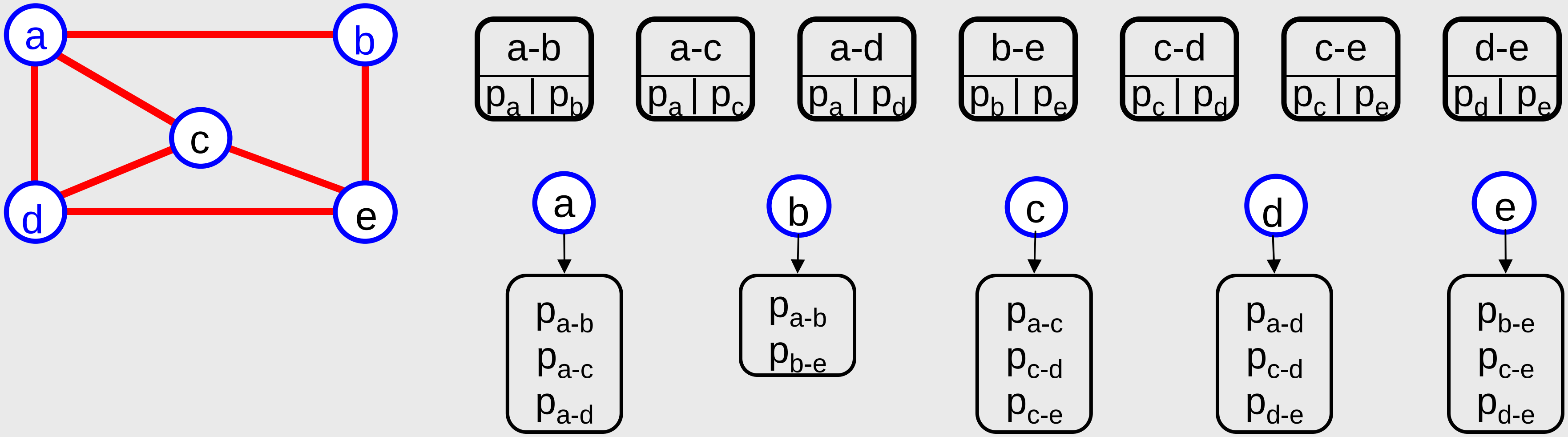
Analysis of DFS

- Setting/getting a vertex/edge label takes ***O***(1) time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- DFS runs in ***O***(***n*** + ***m***) time provided the graph is represented by the adjacency list structure

Depth-First Search

- DFS on a graph with ***n*** vertices and ***m*** edges takes ***O***(***n*** + ***m***) time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph

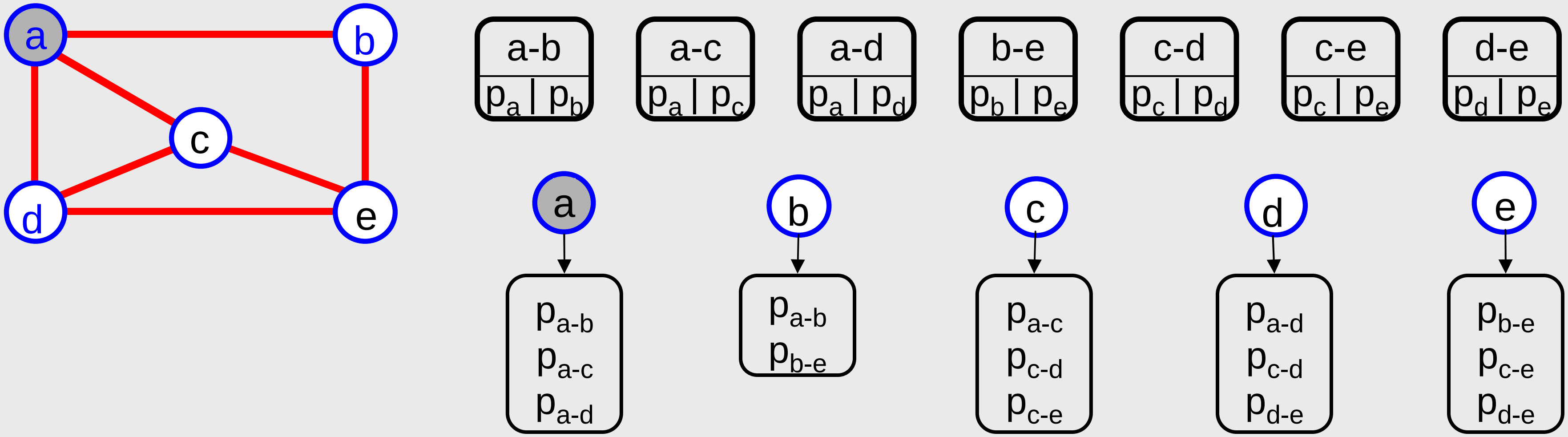
Review: Representation



Space =

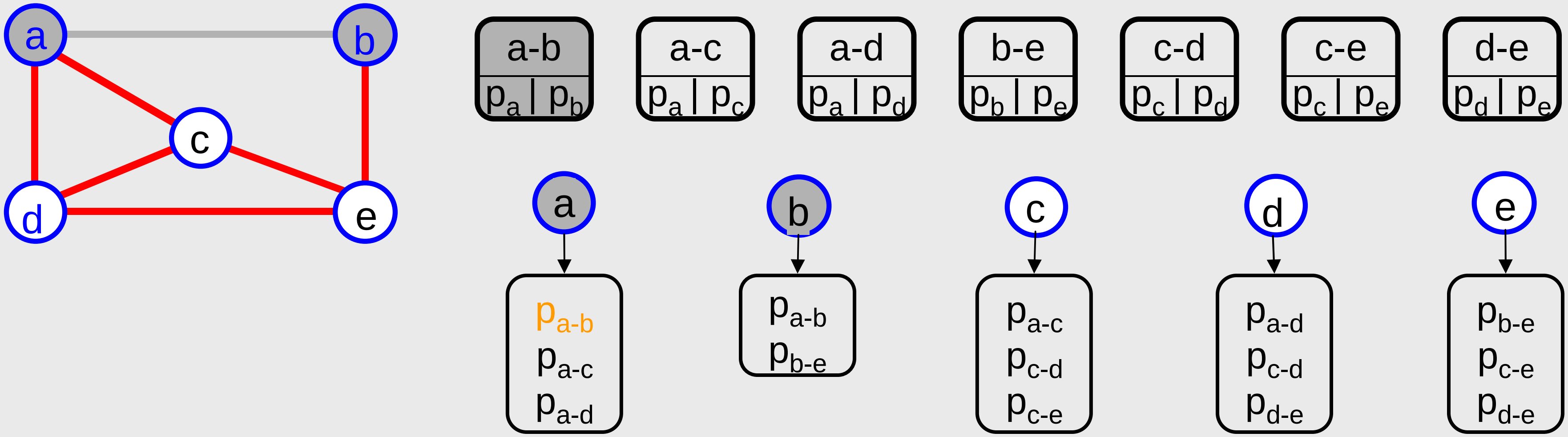
$$\Theta\,(n + \Sigma\, \deg(v)) = \Theta(n + m)$$

Review: DFS



a
b
c
d
e
a-b
p_a | p_b
a-c
p_a | p_c
a-d
p_a | p_d
b-e
p_b | p_e
c-d
p_c | p_d
c-e
p_c | p_e
d-e
p_d | p_e
a
p_{a-b}
p_{a-c}
p_{a-d}
b
p_{a-b}
p_{b-e}
c
p_{a-c}
p_{c-d}
p_{c-e}
d
p_{a-d}
p_{c-d}
p_{d-e}
e
p_{b-e}
p_{c-e}
p_{d-e}

Review: DFS



a-b $p_a \mid p_b$

a-c $p_a \mid p_c$

a-d $p_a \mid p_d$

b-e $p_b \mid p_e$

c-d $p_c \mid p_d$

c-e $p_c \mid p_e$

d-e $p_d \mid p_e$

a: $p_{a\text{-}b}$, $p_{a\text{-}c}$, $p_{a\text{-}d}$

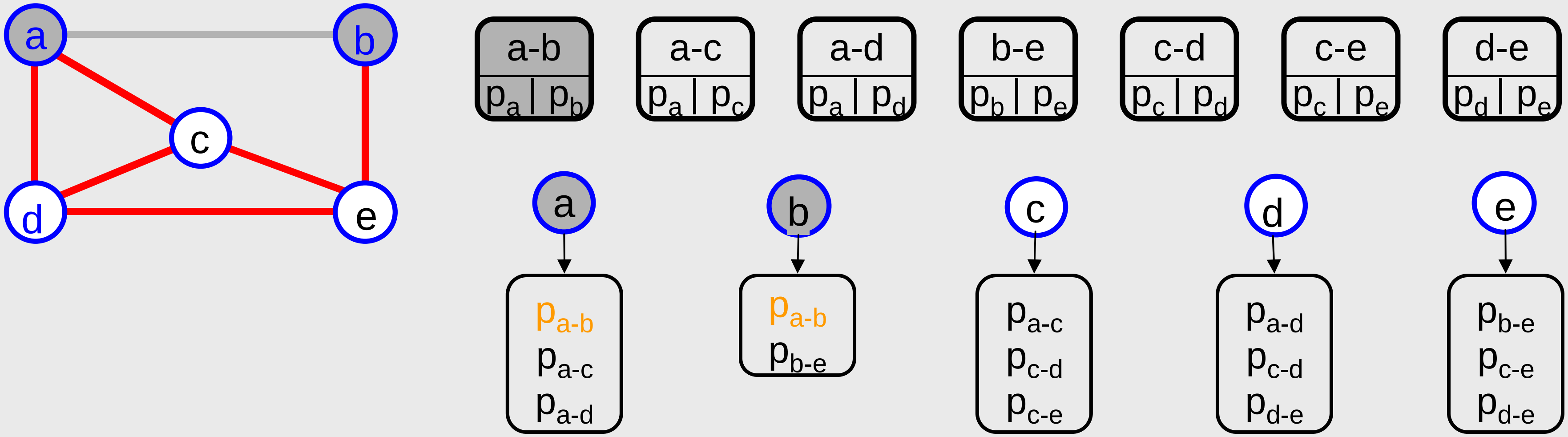
b: $p_{a\text{-}b}$, $p_{b\text{-}e}$

c: $p_{a\text{-}c}$, $p_{c\text{-}d}$, $p_{c\text{-}e}$

d: $p_{a\text{-}d}$, $p_{c\text{-}d}$, $p_{d\text{-}e}$

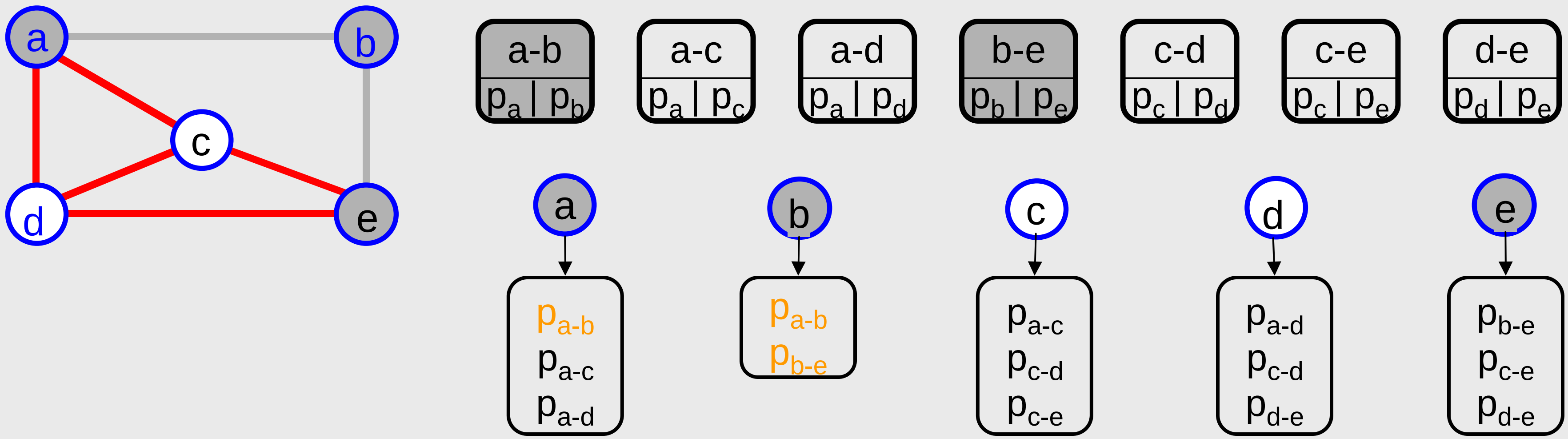
e: $p_{b\text{-}e}$, $p_{c\text{-}e}$, $p_{d\text{-}e}$

Review: DFS



a
b
c
d
e
a-b
p_a | p_b
a-c
p_a | p_c
a-d
p_a | p_d
b-e
p_b | p_e
c-d
p_c | p_d
c-e
p_c | p_e
d-e
p_d | p_e
a
$p_{a\text{-}b}$
$p_{a\text{-}c}$
$p_{a\text{-}d}$
b
$p_{a\text{-}b}$
$p_{b\text{-}e}$
c
$p_{a\text{-}c}$
$p_{c\text{-}d}$
$p_{c\text{-}e}$
d
$p_{a\text{-}d}$
$p_{c\text{-}d}$
$p_{d\text{-}e}$
e
$p_{b\text{-}e}$
$p_{c\text{-}e}$
$p_{d\text{-}e}$

Review: DFS



a-b $p_a \mid p_b$

a-c $p_a \mid p_c$

a-d $p_a \mid p_d$

b-e $p_b \mid p_e$

c-d $p_c \mid p_d$

c-e $p_c \mid p_e$

d-e $p_d \mid p_e$

a: $p_{a\text{-}b}$, $p_{a\text{-}c}$, $p_{a\text{-}d}$

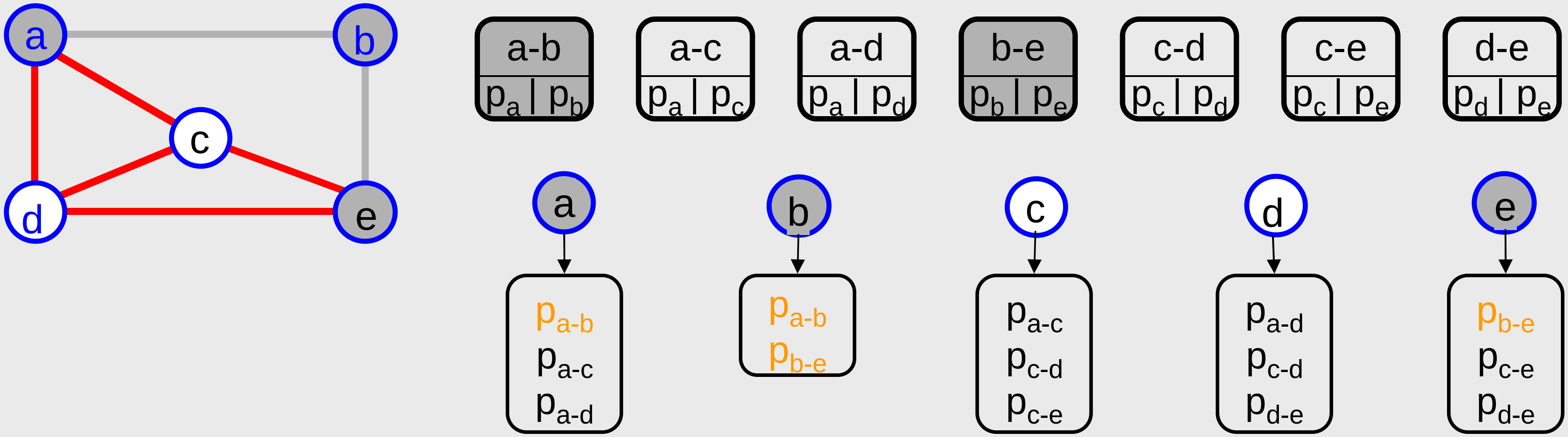
b: $p_{a\text{-}b}$, $p_{b\text{-}e}$

c: $p_{a\text{-}c}$, $p_{c\text{-}d}$, $p_{c\text{-}e}$

d: $p_{a\text{-}d}$, $p_{c\text{-}d}$, $p_{d\text{-}e}$

e: $p_{b\text{-}e}$, $p_{c\text{-}e}$, $p_{d\text{-}e}$

Review: DFS



a-b $p_a \mid p_b$

a-c $p_a \mid p_c$

a-d $p_a \mid p_d$

b-e $p_b \mid p_e$

c-d $p_c \mid p_d$

c-e $p_c \mid p_e$

d-e $p_d \mid p_e$

a: p_{a-b} p_{a-c} p_{a-d}

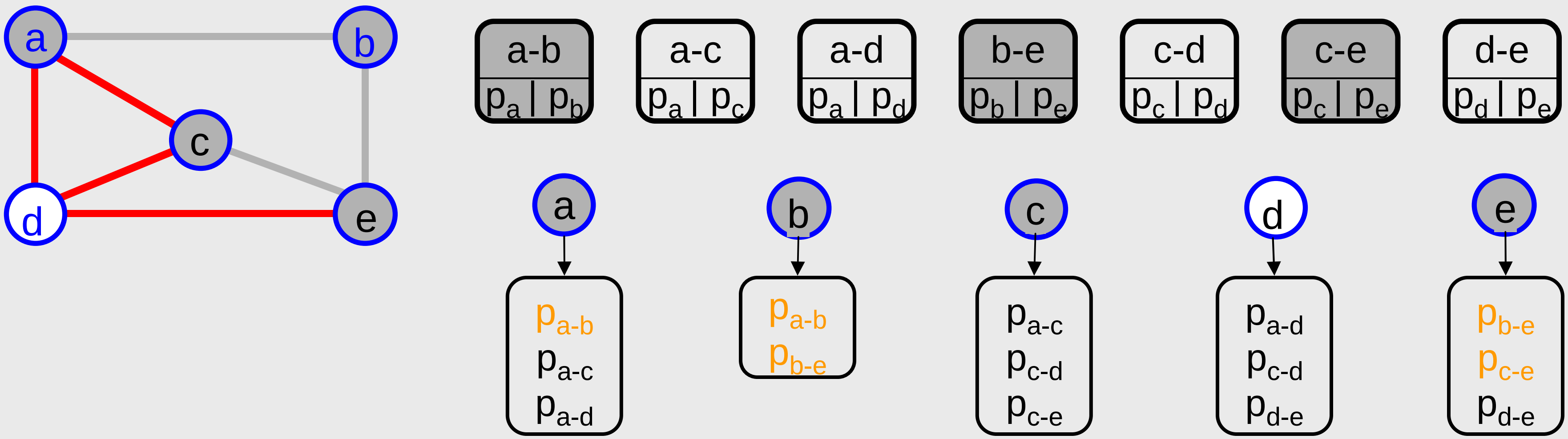
b: p_{a-b} p_{b-e}

c: p_{a-c} p_{c-d} p_{c-e}

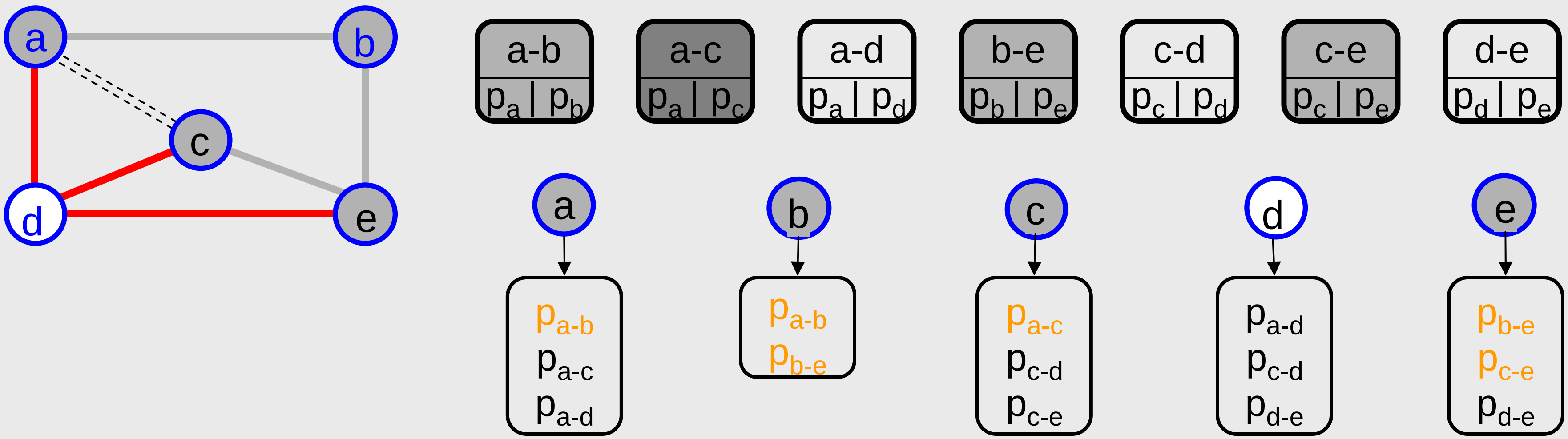
d: p_{a-d} p_{c-d} p_{d-e}

e: p_{b-e} p_{c-e} p_{d-e}

Review: DFS


a
b
c
d
e
a-b
$p_a \mid p_b$
a-c
$p_a \mid p_c$
a-d
$p_a \mid p_d$
b-e
$p_b \mid p_e$
c-d
$p_c \mid p_d$
c-e
$p_c \mid p_e$
d-e
$p_d \mid p_e$
a
p_{a-b}
p_{a-c}
p_{a-d}
b
p_{a-b}
p_{b-e}
c
p_{a-c}
p_{c-d}
p_{c-e}
d
p_{a-d}
p_{c-d}
p_{d-e}
e
p_{b-e}
p_{c-e}
p_{d-e}

Review: DFS



a-b $p_a | p_b$

a-c $p_a | p_c$

a-d $p_a | p_d$

b-e $p_b | p_e$

c-d $p_c | p_d$

c-e $p_c | p_e$

d-e $p_d | p_e$

a: $p_{a\text{-}b}$ $p_{a\text{-}c}$ $p_{a\text{-}d}$

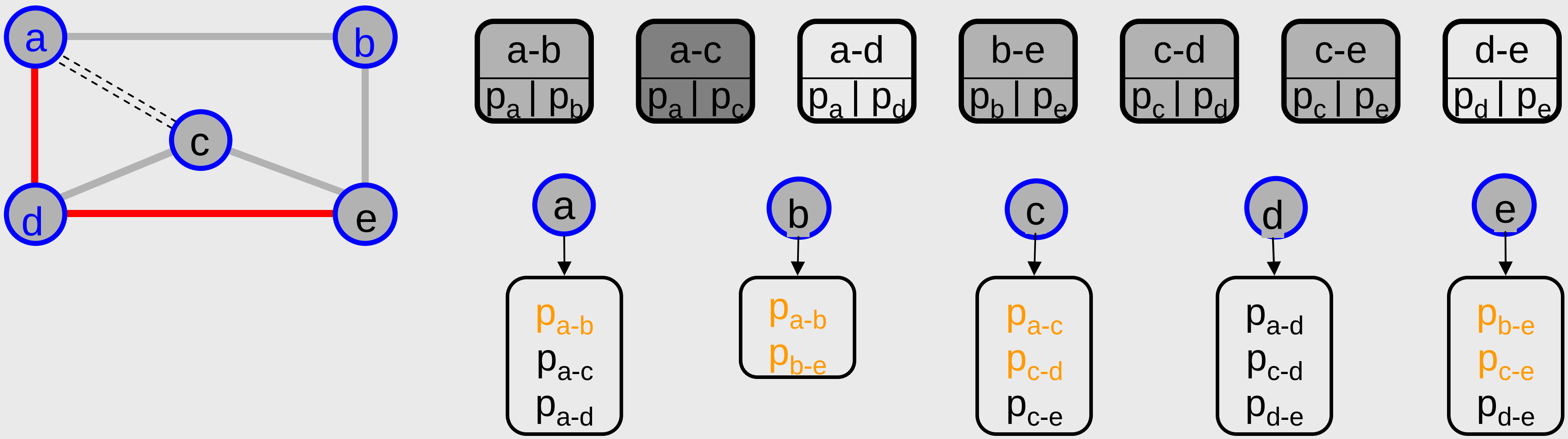
b: $p_{a\text{-}b}$ $p_{b\text{-}e}$

c: $p_{a\text{-}c}$ $p_{c\text{-}d}$ $p_{c\text{-}e}$

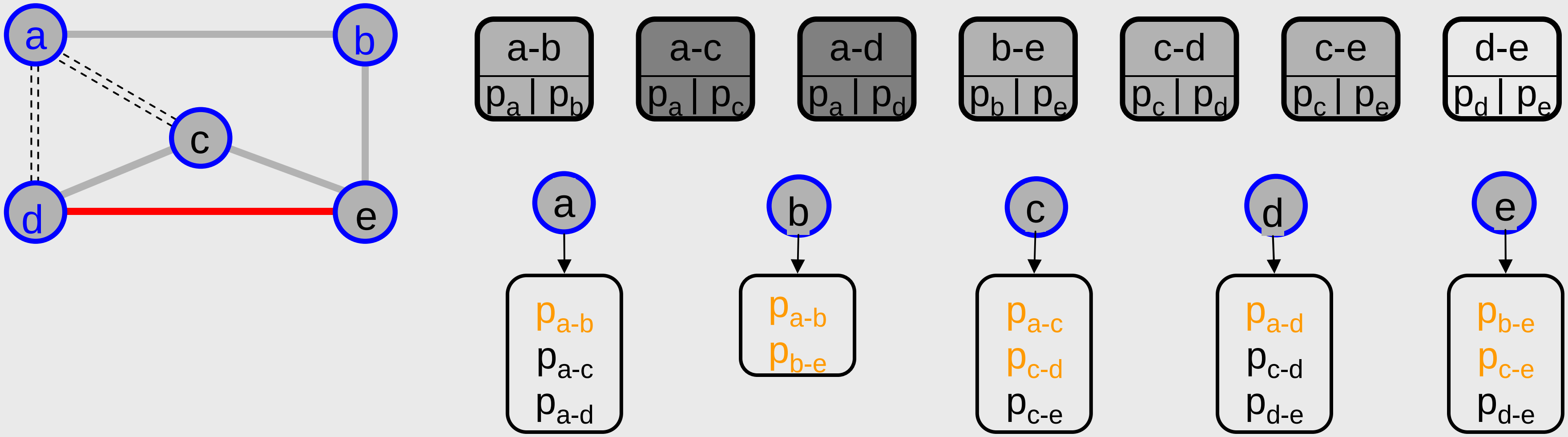
d: $p_{a\text{-}d}$ $p_{c\text{-}d}$ $p_{d\text{-}e}$

e: $p_{b\text{-}e}$ $p_{c\text{-}e}$ $p_{d\text{-}e}$

Review: DFS


a
b
c
d
e
a-b
p_a | p_b
a-c
p_a | p_c
a-d
p_a | p_d
b-e
p_b | p_e
c-d
p_c | p_d
c-e
p_c | p_e
d-e
p_d | p_e
a
p_a-b
p_a-c
p_a-d
b
p_a-b
p_b-e
c
p_a-c
p_c-d
p_c-e
d
p_a-d
p_c-d
p_d-e
e
p_b-e
p_c-e
p_d-e

Review: DFS



a-b $p_a | p_b$

a-c $p_a | p_c$

a-d $p_a | p_d$

b-e $p_b | p_e$

c-d $p_c | p_d$

c-e $p_c | p_e$

d-e $p_d | p_e$

a: p_{a-b}, p_{a-c}, p_{a-d}

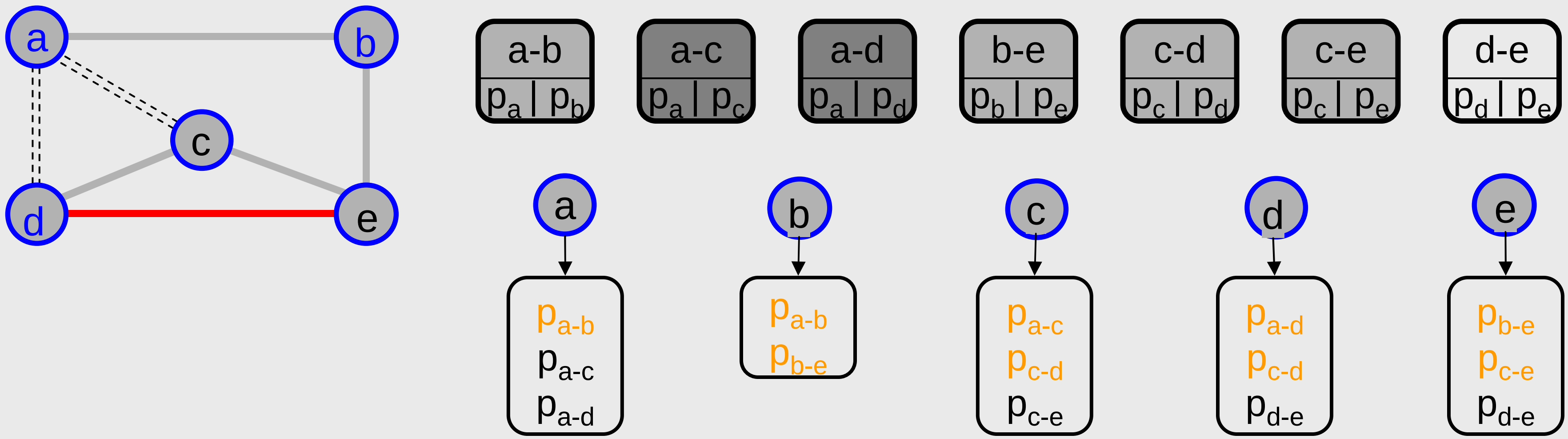
b: p_{a-b}, p_{b-e}

c: p_{a-c}, p_{c-d}, p_{c-e}

d: p_{a-d}, p_{c-d}, p_{d-e}

e: p_{b-e}, p_{c-e}, p_{d-e}

Review: DFS



a-b $p_a \mid p_b$

a-c $p_a \mid p_c$

a-d $p_a \mid p_d$

b-e $p_b \mid p_e$

c-d $p_c \mid p_d$

c-e $p_c \mid p_e$

d-e $p_d \mid p_e$

a: $p_{a\text{-}b}$, $p_{a\text{-}c}$, $p_{a\text{-}d}$

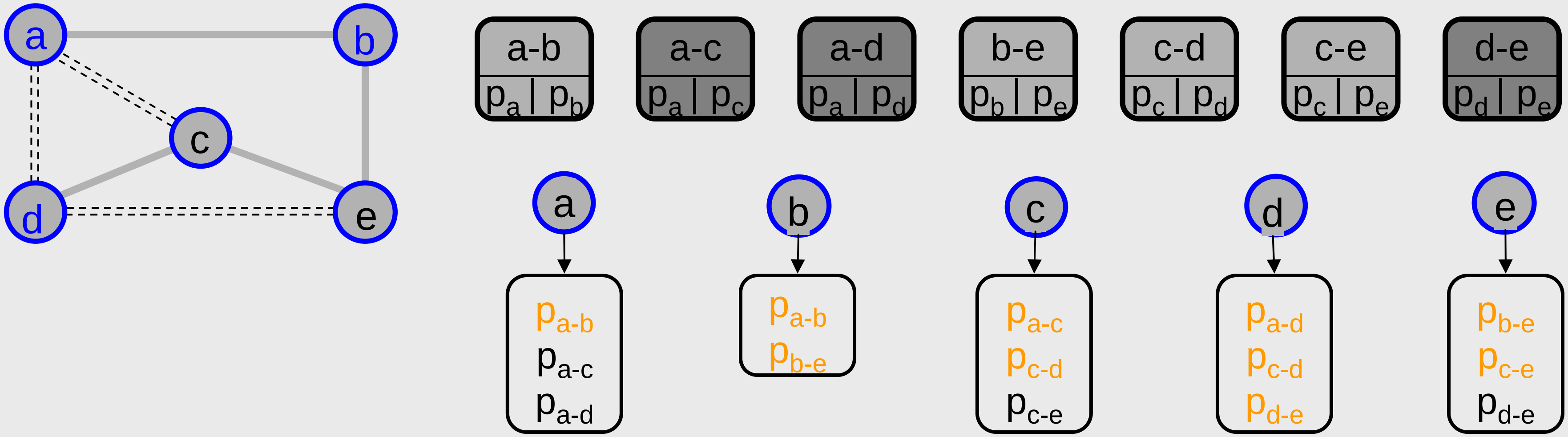
b: $p_{a\text{-}b}$, $p_{b\text{-}e}$

c: $p_{a\text{-}c}$, $p_{c\text{-}d}$, $p_{c\text{-}e}$

d: $p_{a\text{-}d}$, $p_{c\text{-}d}$, $p_{d\text{-}e}$

e: $p_{b\text{-}e}$, $p_{c\text{-}e}$, $p_{d\text{-}e}$

Review: DFS



a-b p_a | p_b

a-c p_a | p_c

a-d p_a | p_d

b-e p_b | p_e

c-d p_c | p_d

c-e p_c | p_e

d-e p_d | p_e

a: p_{a-b} p_{a-c} p_{a-d}

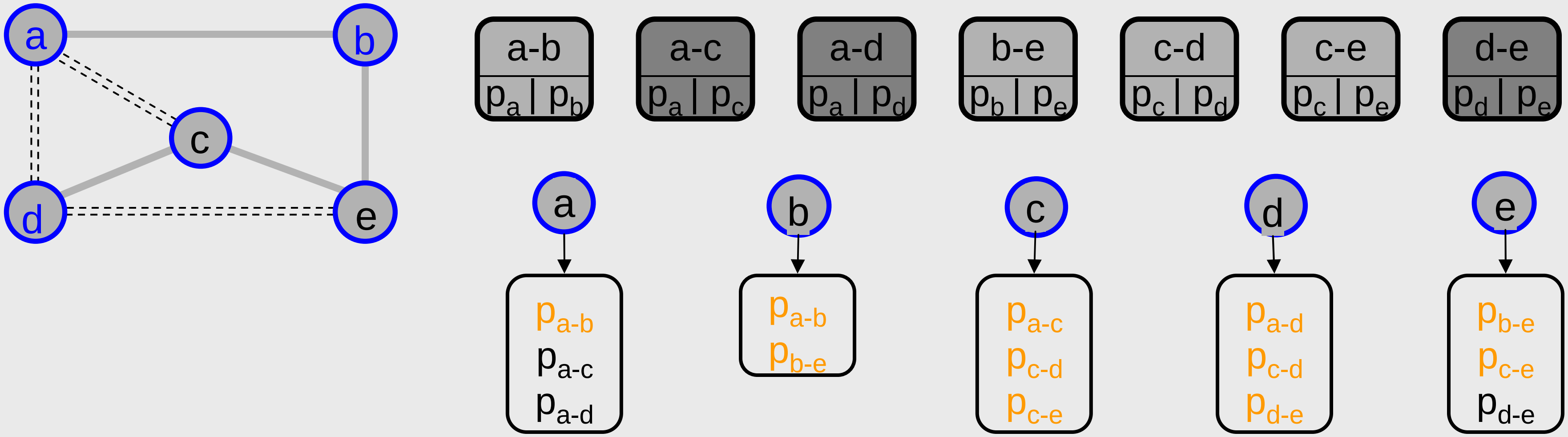
b: p_{a-b} p_{b-e}

c: p_{a-c} p_{c-d} p_{c-e}

d: p_{a-d} p_{c-d} p_{d-e}

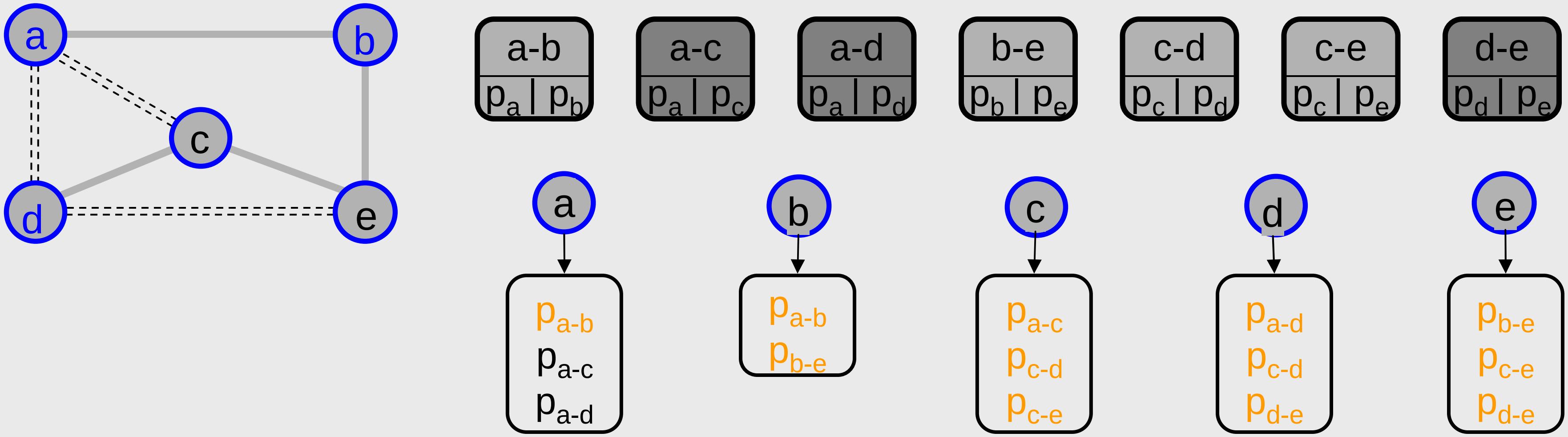
e: p_{b-e} p_{c-e} p_{d-e}

Review: DFS

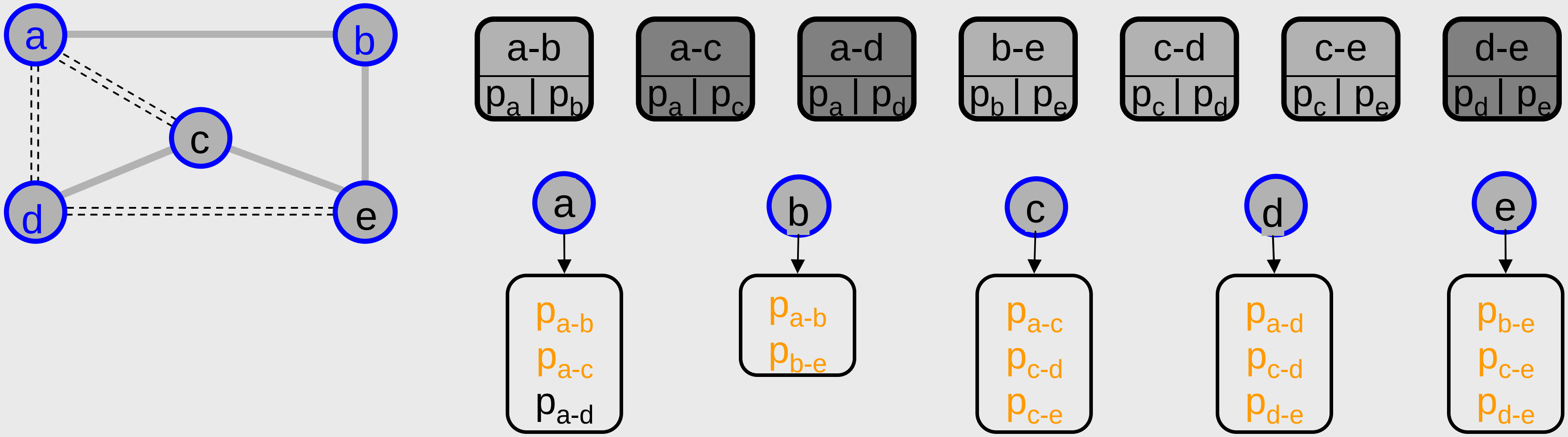


a
b
c
d
e
a-b
$p_a | p_b$
a-c
$p_a | p_c$
a-d
$p_a | p_d$
b-e
$p_b | p_e$
c-d
$p_c | p_d$
c-e
$p_c | p_e$
d-e
$p_d | p_e$
a
p_{a-b}
p_{a-c}
p_{a-d}
b
p_{a-b}
p_{b-e}
c
p_{a-c}
p_{c-d}
p_{c-e}
d
p_{a-d}
p_{c-d}
p_{d-e}
e
p_{b-e}
p_{c-e}
p_{d-e}

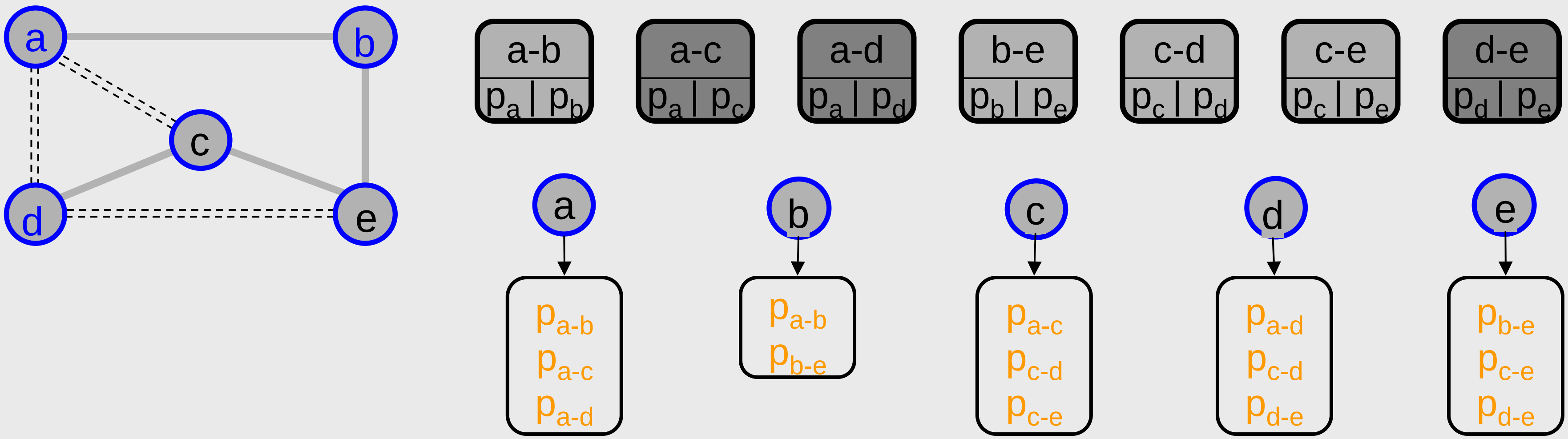
Review: DFS


a
b
c
d
e
a-b
$p_a \mid p_b$
a-c
$p_a \mid p_c$
a-d
$p_a \mid p_d$
b-e
$p_b \mid p_e$
c-d
$p_c \mid p_d$
c-e
$p_c \mid p_e$
d-e
$p_d \mid p_e$
a
p_{a-b}
p_{a-c}
p_{a-d}
b
p_{a-b}
p_{b-e}
c
p_{a-c}
p_{c-d}
p_{c-e}
d
p_{a-d}
p_{c-d}
p_{d-e}
e
p_{b-e}
p_{c-e}
p_{d-e}

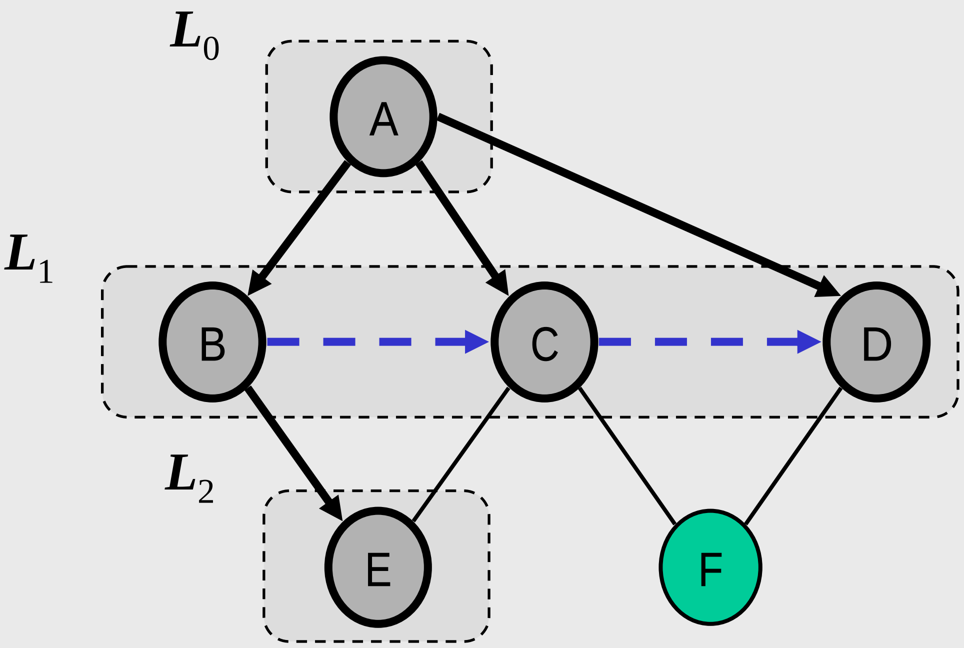
Review: DFS


a
b
c
d
e
a-b
p_a | p_b
a-c
p_a | p_c
a-d
p_a | p_d
b-e
p_b | p_e
c-d
p_c | p_d
c-e
p_c | p_e
d-e
p_d | p_e
a
p_{a-b}
p_{a-c}
p_{a-d}
b
p_{a-b}
p_{b-e}
c
p_{a-c}
p_{c-d}
p_{c-e}
d
p_{a-d}
p_{c-d}
p_{d-e}
e
p_{b-e}
p_{c-e}
p_{d-e}

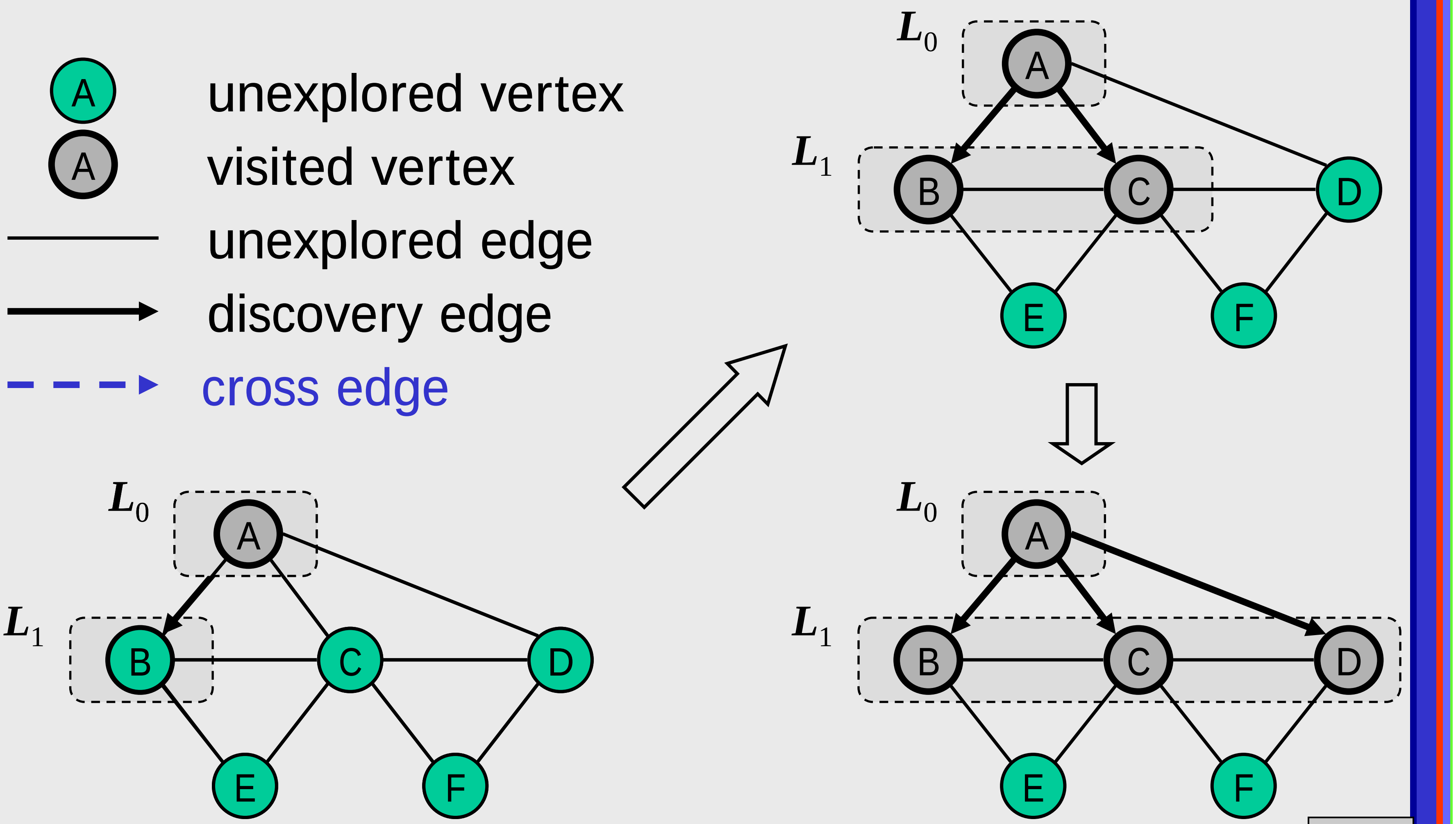
Review: DFS


a
b
c
d
e
a-b
p_a | p_b
a-c
p_a | p_c
a-d
p_a | p_d
b-e
p_b | p_e
c-d
p_c | p_d
c-e
p_c | p_e
d-e
p_d | p_e
a
p_{a-b}
p_{a-c}
p_{a-d}
b
p_{a-b}
p_{b-e}
c
p_{a-c}
p_{c-d}
p_{c-e}
d
p_{a-d}
p_{c-d}
p_{d-e}
e
p_{b-e}
p_{c-e}
p_{d-e}

Breadth-First Search



Example



A unexplored vertex

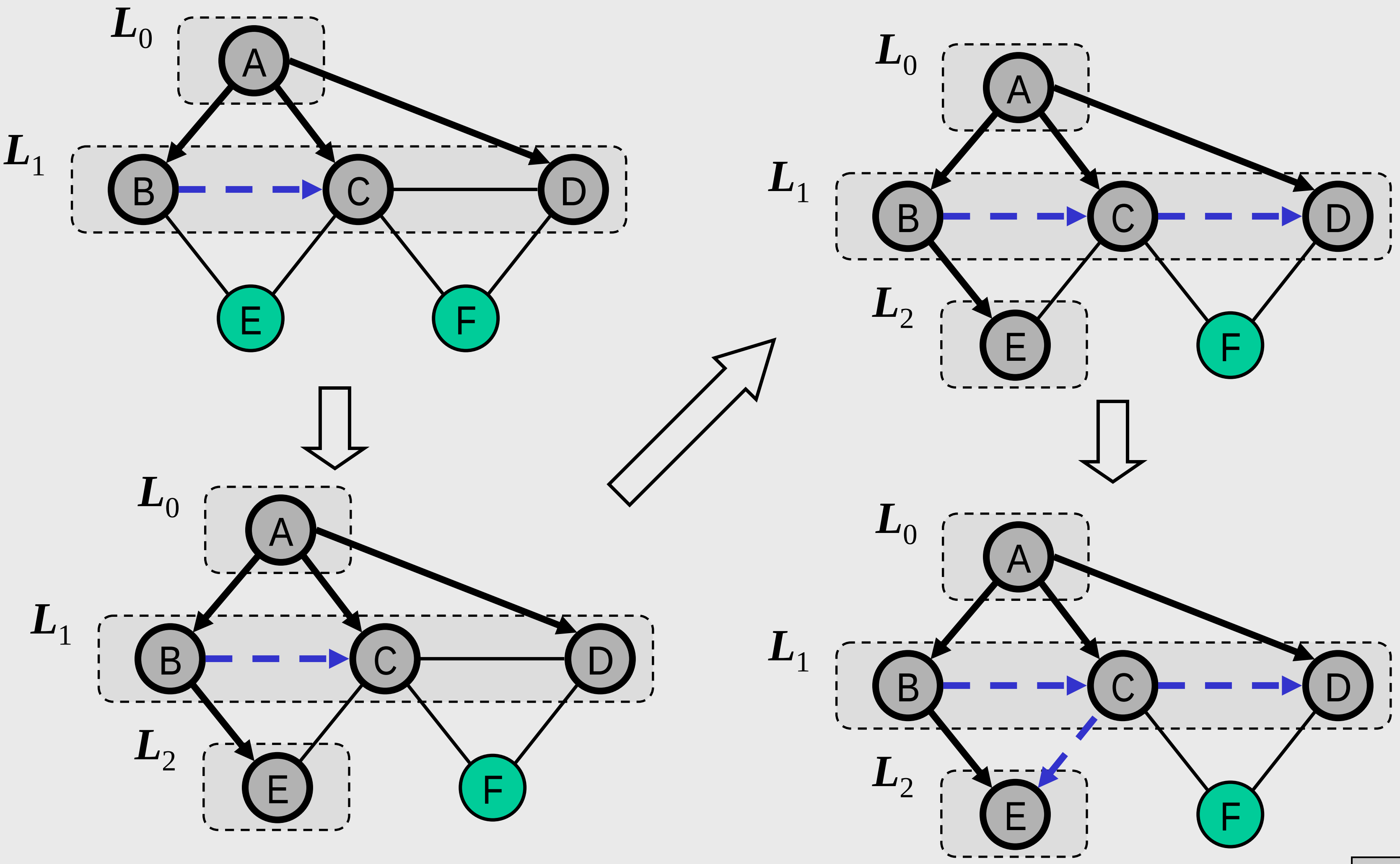
A visited vertex

— unexplored edge

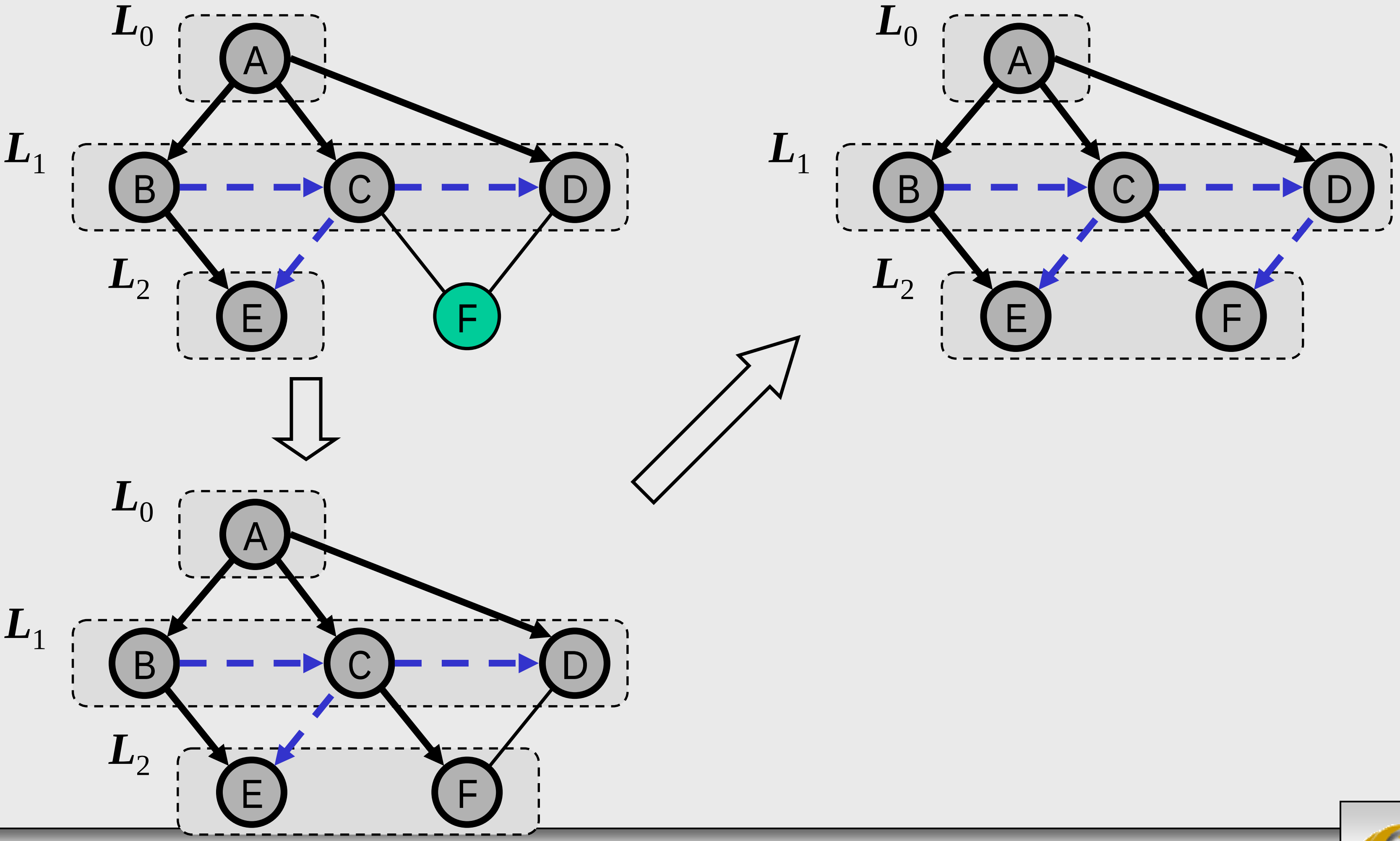
→ discovery edge

- - -▸ cross edge

Example (cont.)



Example (cont.)



BFS Algorithm

- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

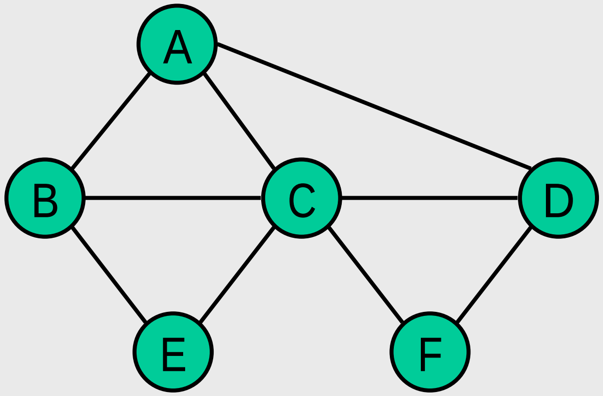
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all $u \in$ G.vertices()
    setLabel(u, UNEXPLORED)
  for all $e \in$ G.edges()
    setLabel(e, UNEXPLORED)
  for all $v \in$ G.vertices()
    if getLabel(v) = UNEXPLORED
        BFS(G, v)
```

```
Algorithm BFS(G, s)
  L $\leftarrow$ new empty queue
  L.enqueue(s)
  setLabel(s, VISITED)
  while $\neg$L.isEmpty()
      v $\leftarrow$ L.dequeue()
      for all $e \in$ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w $\leftarrow$ opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            L.enqueue(w)
          else
            setLabel(e, CROSS)
```

Properties

Notation

G_s: connected component of s

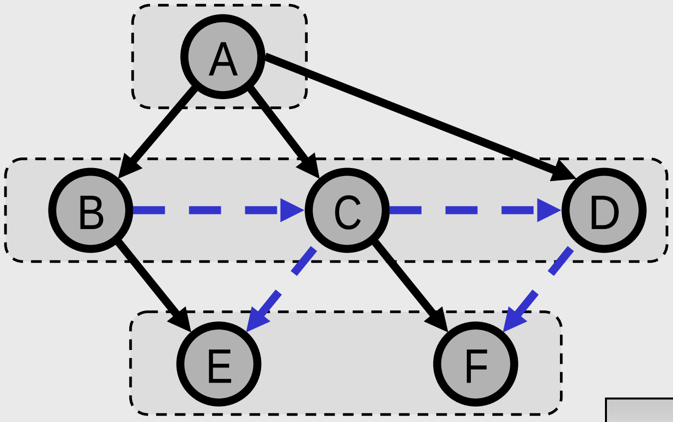


Property 1

BFS(***G, s***) visits all the vertices and edges of G_s

Property 2

The discovery edges labeled by ***BFS***(***G, s***) form a spanning tree T_s of G_s



Analysis

- Setting/getting a vertex/edge label takes ***O***(1) time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a queue ***L***
- Method incidentEdges is called once for each vertex
- BFS runs in ***O***(***n*** + ***m***) time provided the graph is represented by the adjacency list structure

Applications

- We can specialize the BFS traversal of a graph ***G*** to solve the following problems in ***O***(***n*** + ***m***) time
  - Compute the connected components of ***G***
  - Compute a spanning forest of ***G***
  - Find a simple cycle in ***G***, or report that ***G*** is a forest
  - Given two vertices of ***G***, find a path in ***G*** between them with the minimum number of edges, or report that no such path exists

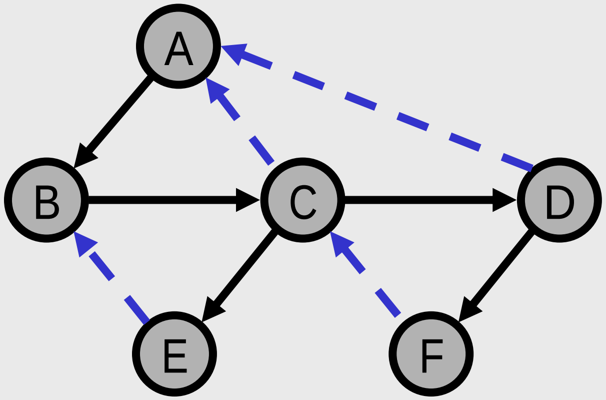
DFS vs. BFS

Back edge (***v***,***w***)

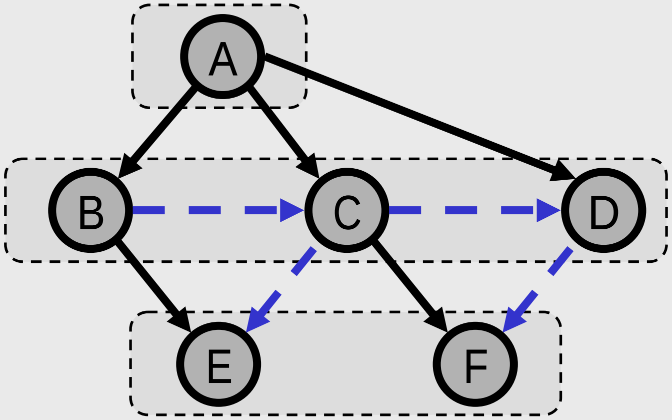
- ***w*** is an ancestor of ***v*** in the tree of discovery edges

Cross edge (***v***,***w***)

- ***w*** is in the same level as ***v*** or in the next level in the tree of discovery edges



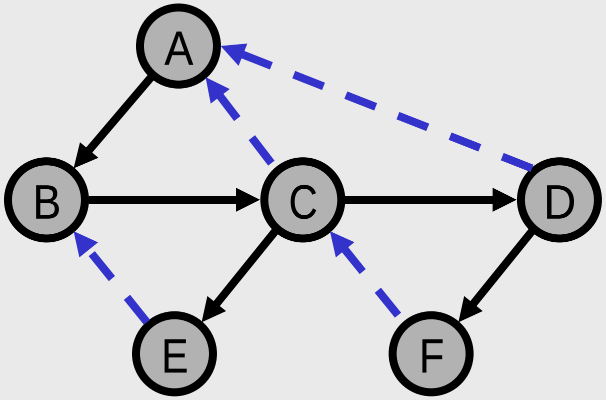
DFS



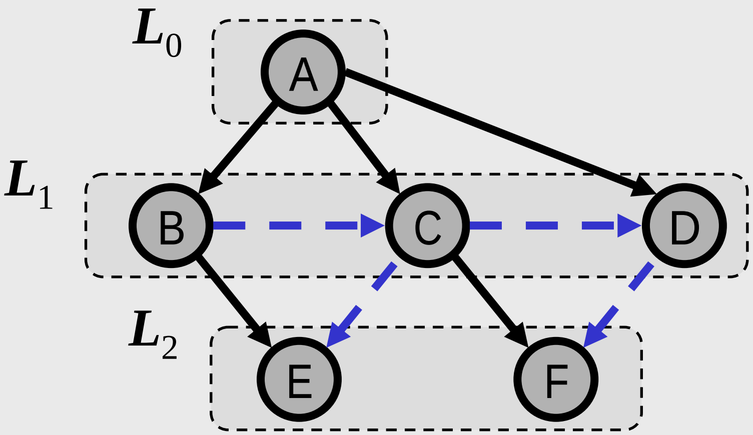
BFS

DFS vs. BFS (cont.)

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



DFS



BFS

Path Finding

- We call ***DFS*(*G, u*)** with ***u*** as the start vertex
- We use a stack ***S*** to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex ***v*** is encountered, we return the path as the contents of the stack

Path Finding

```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if $v = z$
    return S.elements()
  for all $e \in$ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      $w \leftarrow$ opposite(v,e)
      if getLabel(w) = UNEXPLORED
         setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
         setLabel(e, BACK)
  S.pop(v)
```

Cycle Finding

- We use a stack ***S*** to keep track of the path between the start vertex and the current vertex

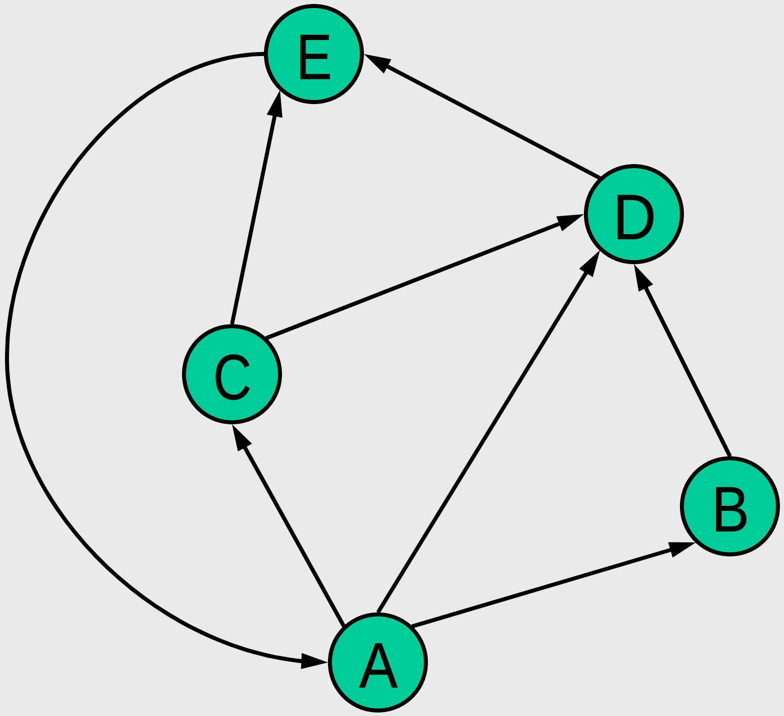
- As soon as a back edge (***v***, ***w***) is encountered, we return the cycle as the portion of the stack from the top to vertex ***w***

Cycle Finding

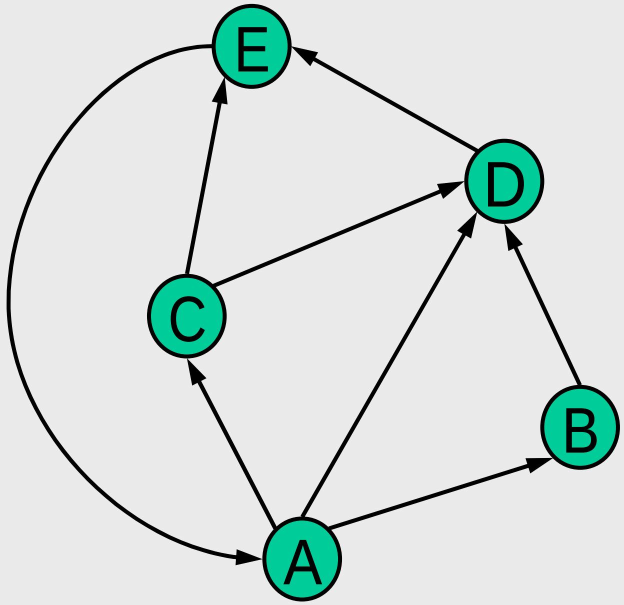
```
Algorithm cycleDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  for all  e ∈ G.incidentEdges(v)
    if  getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      S.push(e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        pathDFS(G, w, z)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```

Digraphs

- A **digraph** is a graph whose edges are all directed
  - Short for “directed graph”
- Applications
  - one-way streets
  - flights
  - task scheduling



Digraph Properties



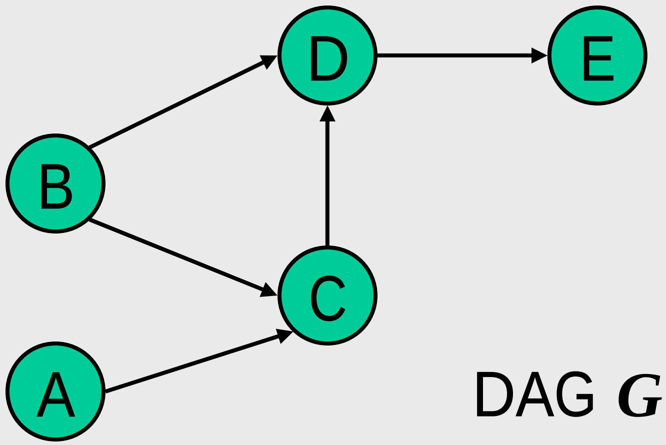
- A graph G=(V,E) such that
  - Each edge goes in one direction:
    - Edge (a,b) goes from a to b, but not b to a.
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of in-edges and out-edges in time proportional to their size.

DAGs and Topological Ordering

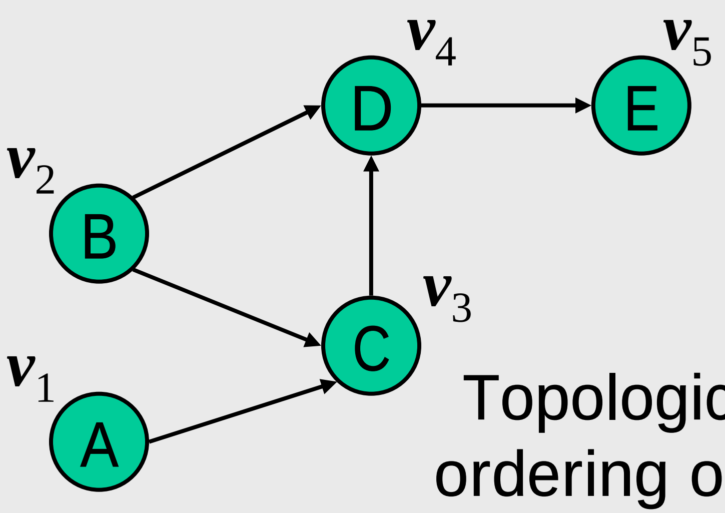
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

  $$\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$$

  of the vertices such that for every edge $(\boldsymbol{v}_i, \boldsymbol{v}_j)$, we have $\boldsymbol{i} < \boldsymbol{j}$
- A digraph admits a topological ordering if and only if it is a DAG



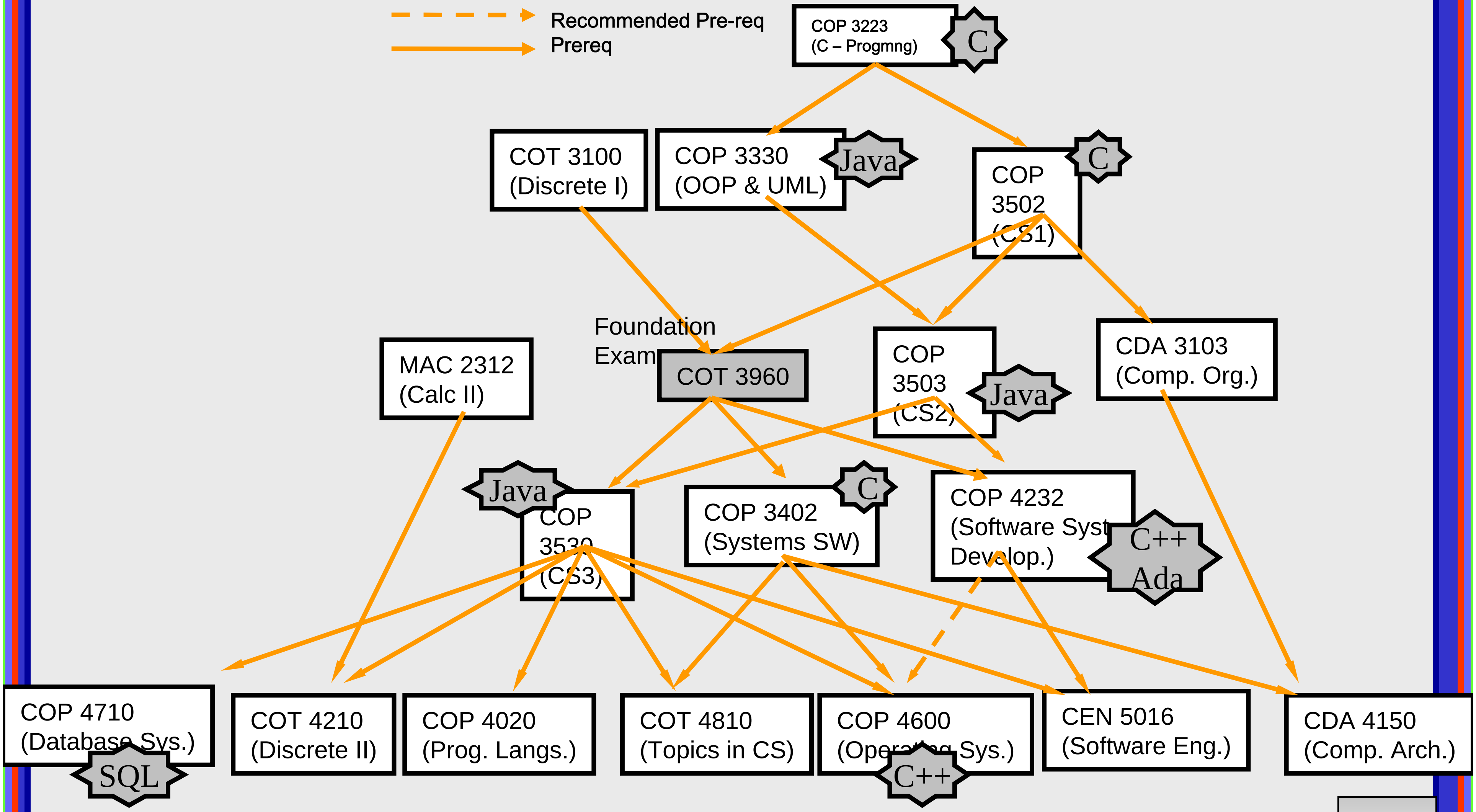
DAG ***G***



Topological ordering of ***G***

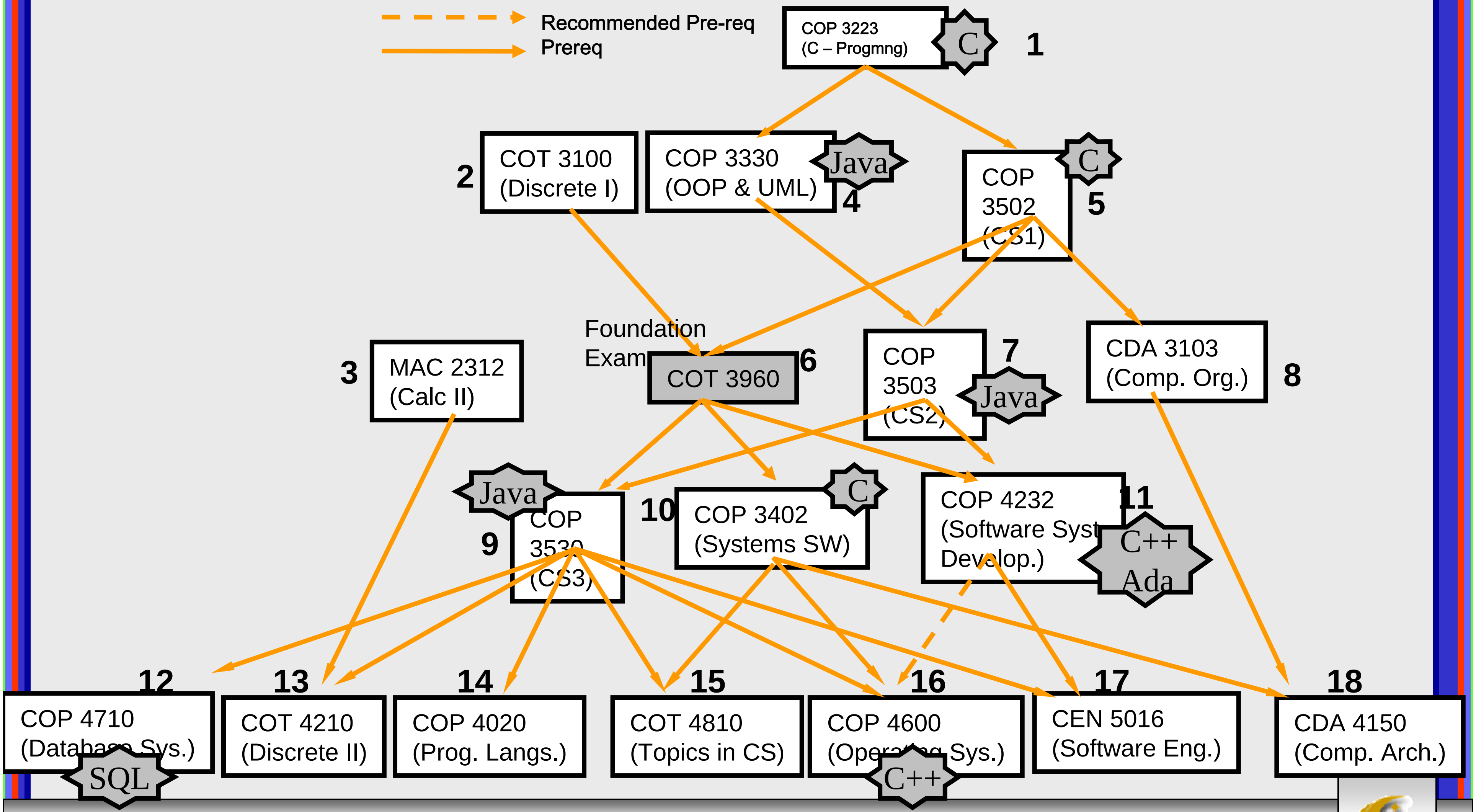
Computer Science Pre-requisite Graph

- edge (a,b) means task a must be completed before b can be started



Topological Sorting

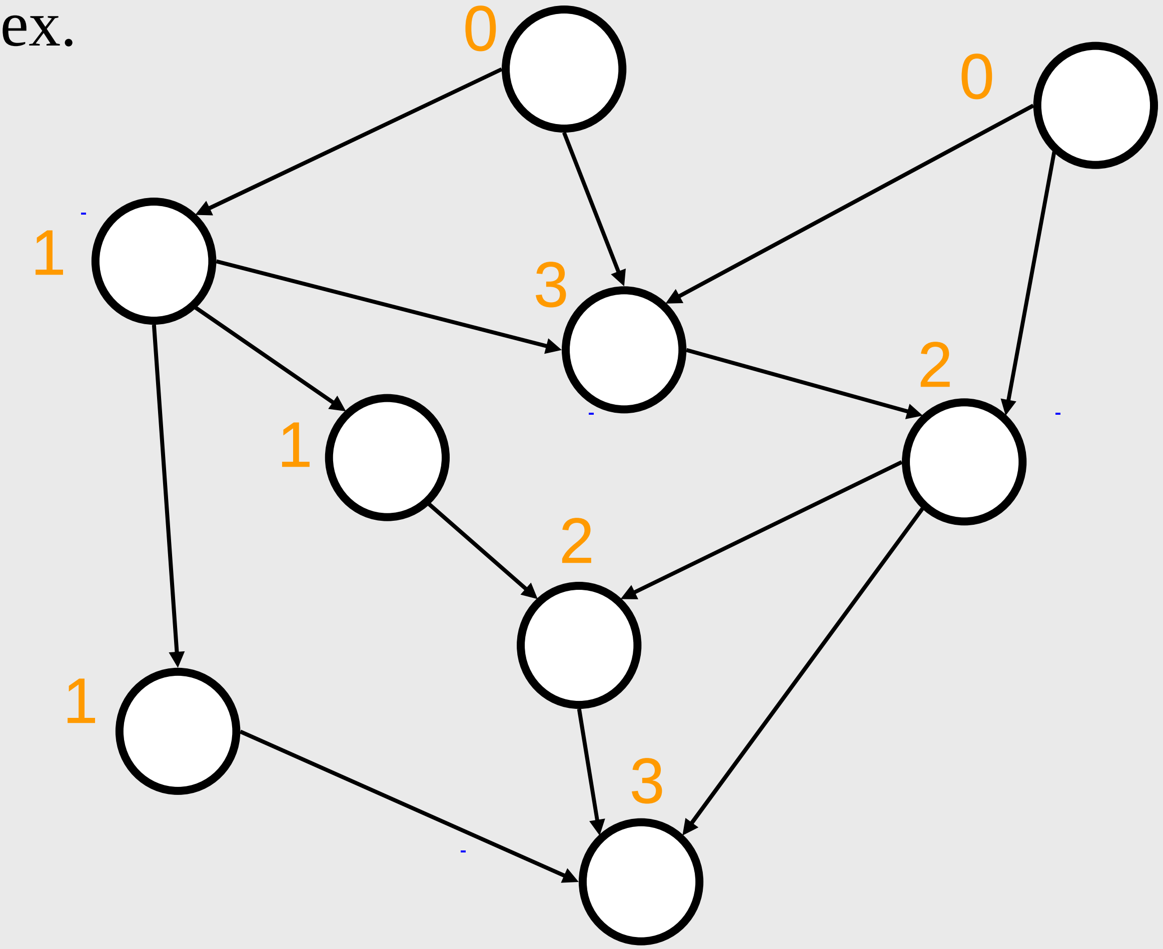
- Number vertices, so that (u,v) in E implies u < v



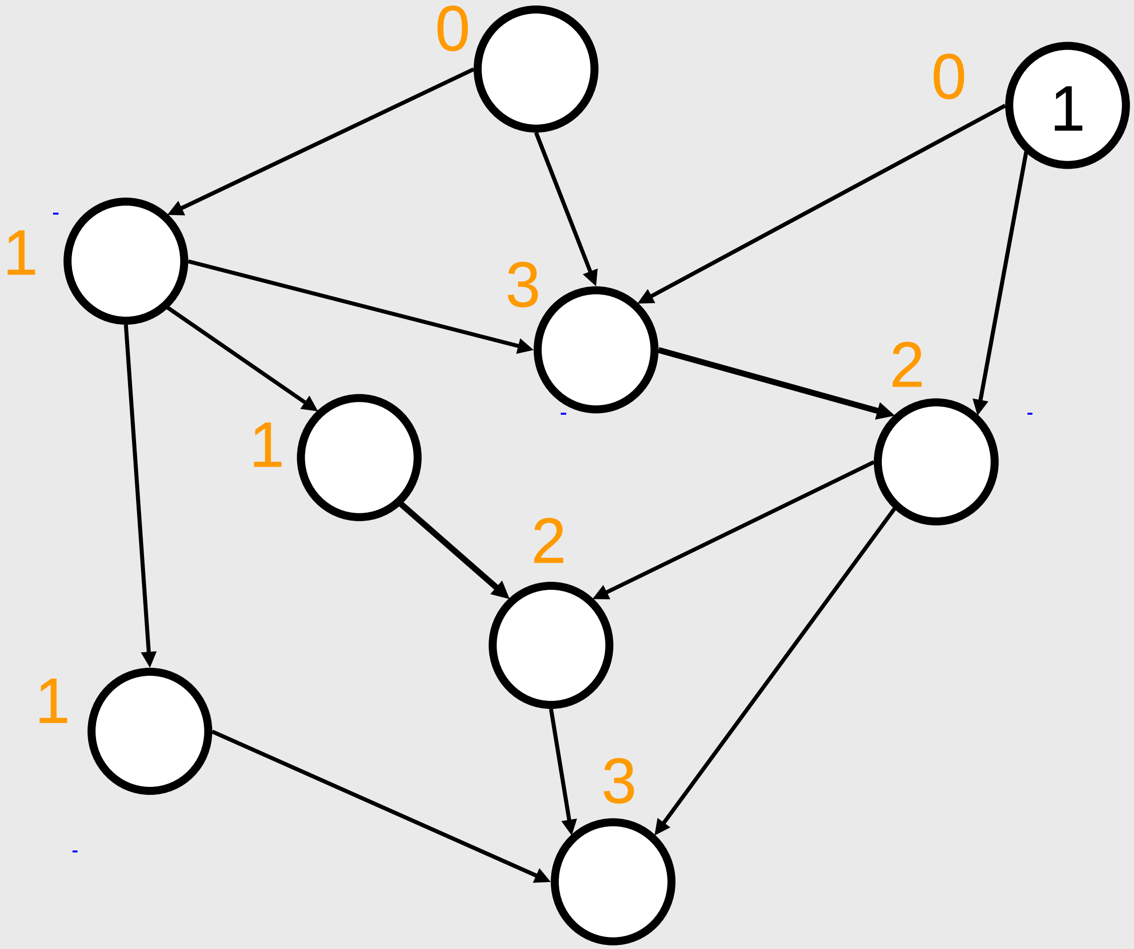
Recommended Pre-req
Prereq
COP 3223 (C – Progmng) C 1
2 COT 3100 (Discrete I)
COP 3330 (OOP & UML) Java 4
COP 3502 (CS1) C 5
3 MAC 2312 (Calc II)
Foundation Exam COT 3960 6
COP 3503 (CS2) 7 Java
CDA 3103 (Comp. Org.) 8
Java COP 3530 (CS3) 9
10 COP 3402 (Systems SW) C
COP 4232 (Software Syst Develop.) 11 C++ Ada
12 COP 4710 (Database Sys.) SQL
13 COT 4210 (Discrete II)
14 COP 4020 (Prog. Langs.)
15 COT 4810 (Topics in CS)
16 COP 4600 (Operating Sys.) C++
17 CEN 5016 (Software Eng.)
18 CDA 4150 (Comp. Arch.)

Topological Sorting Example

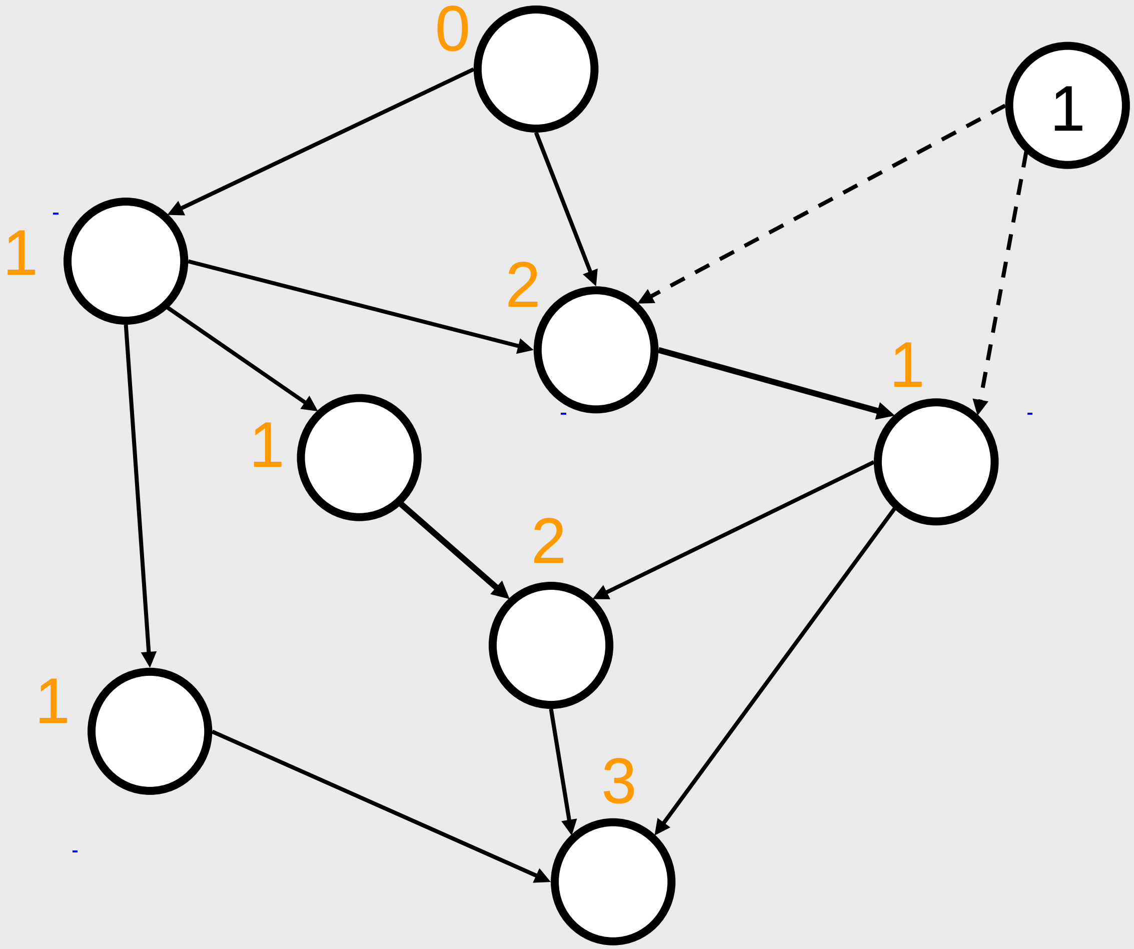
Graph with # of Incident edges for each vertex.



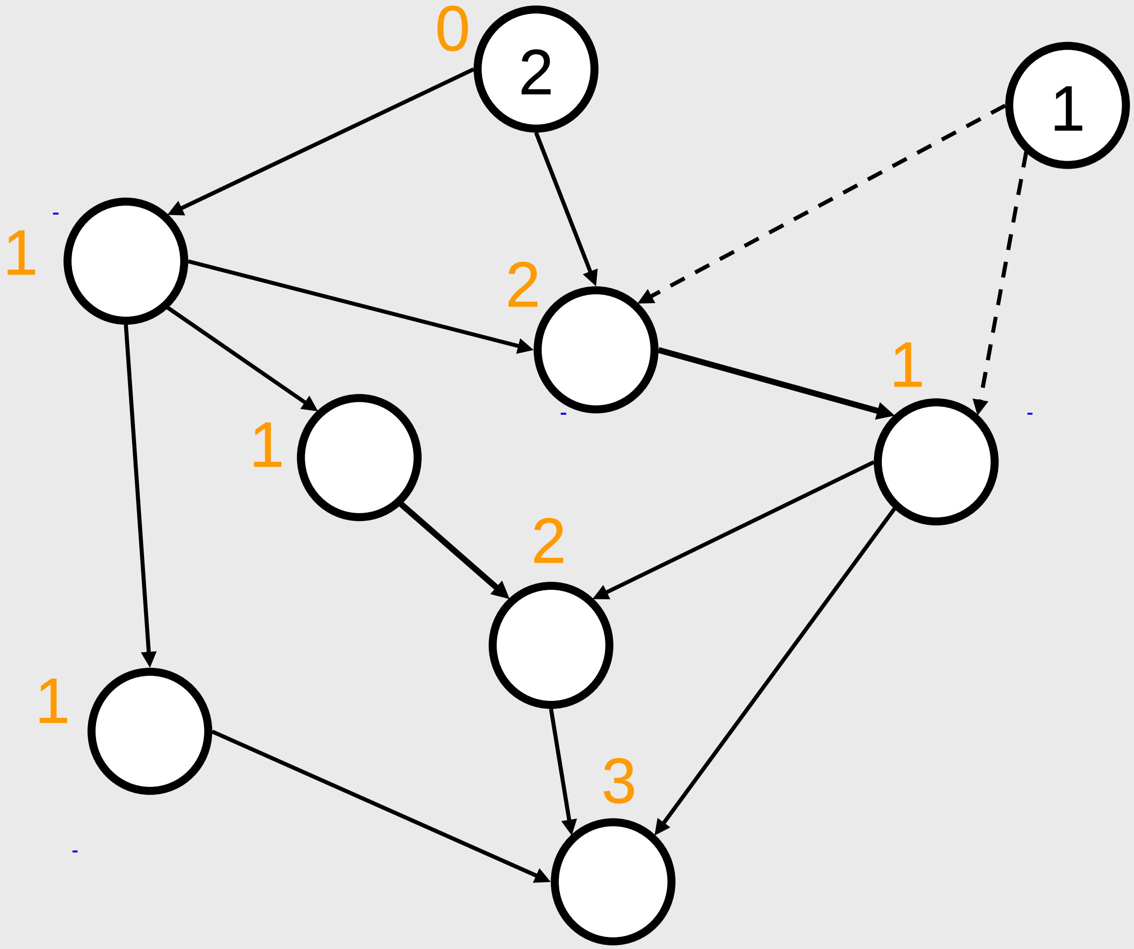
Topological Sorting Example



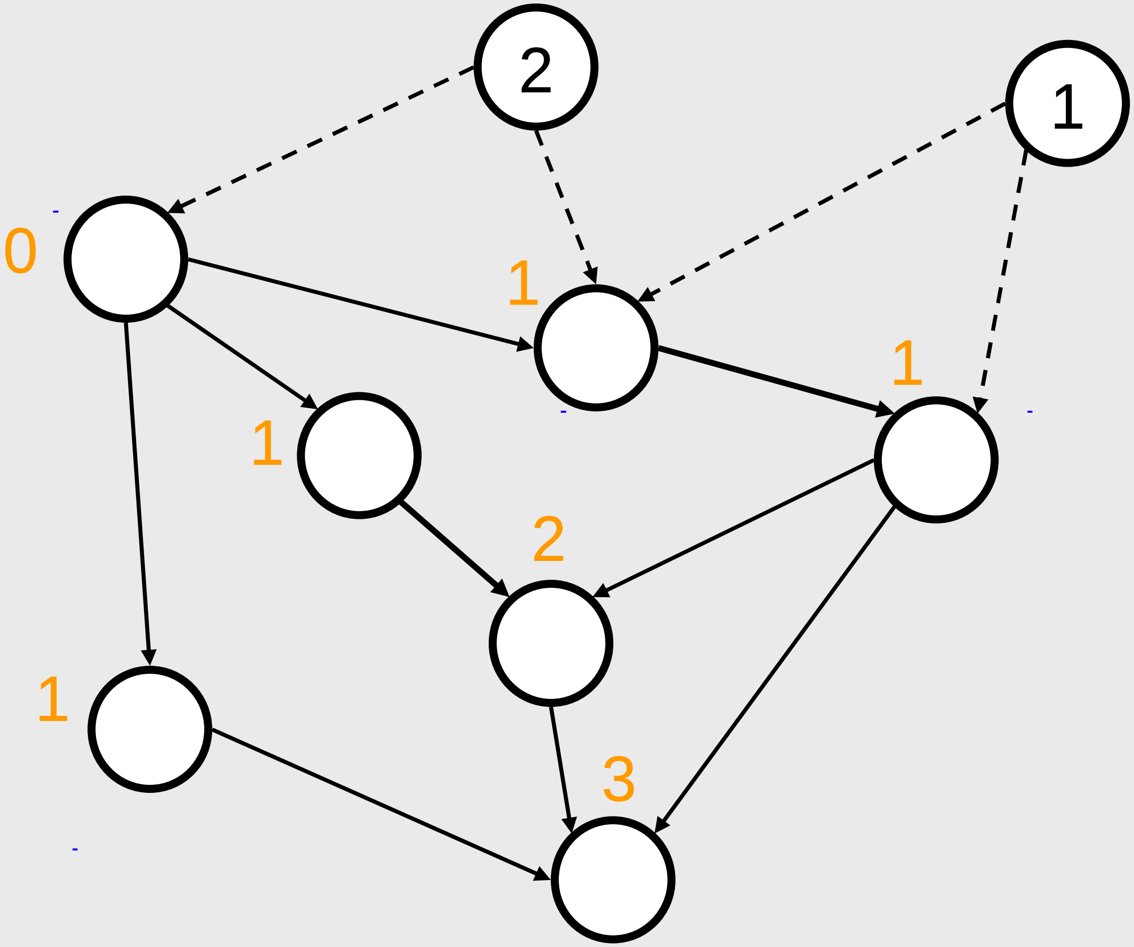
Topological Sorting Example



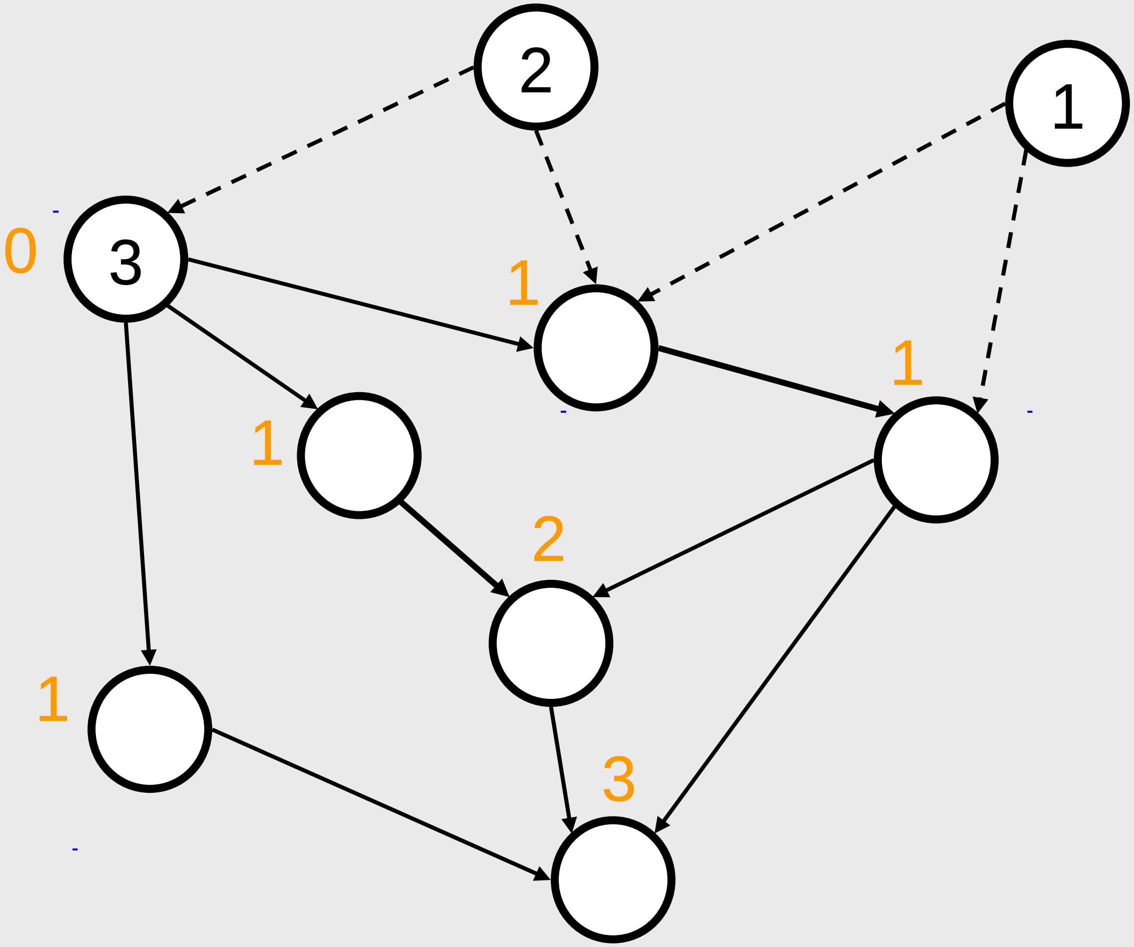
Topological Sorting Example



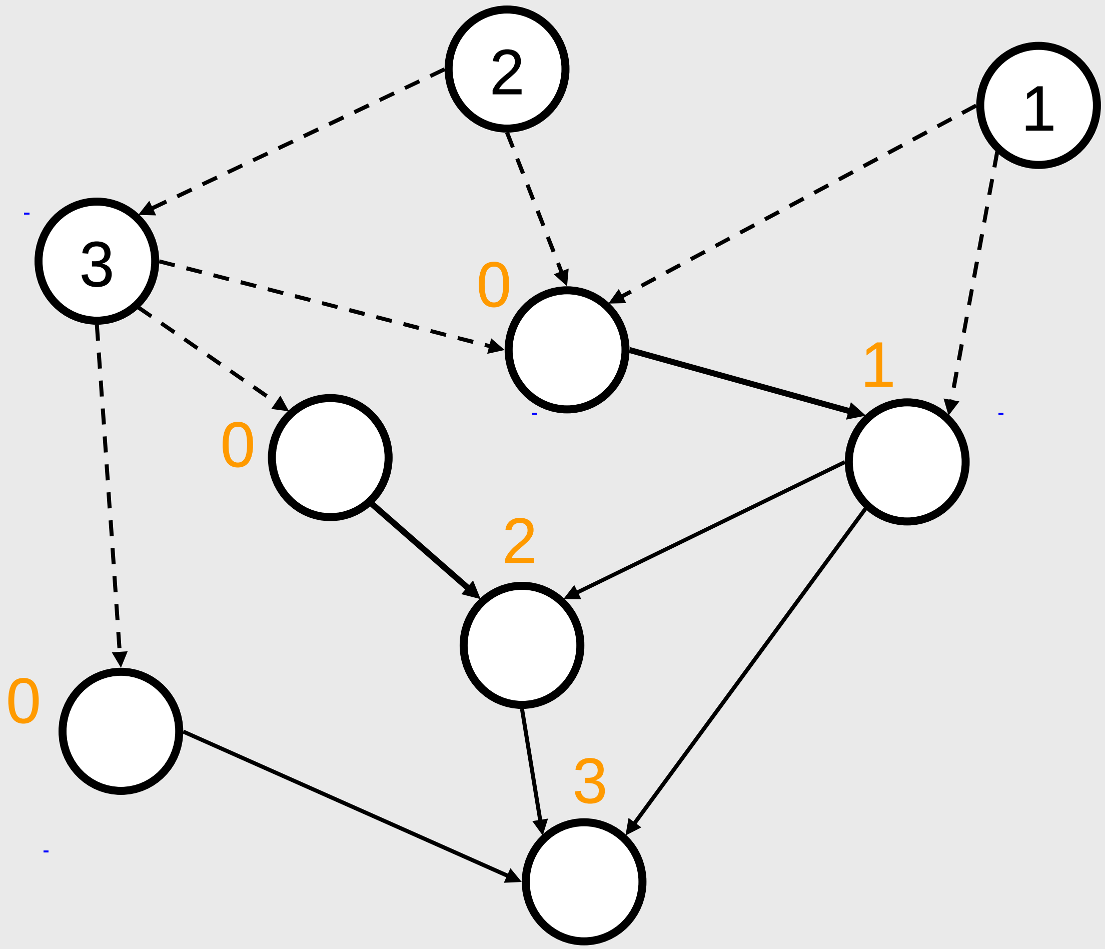
Topological Sorting Example



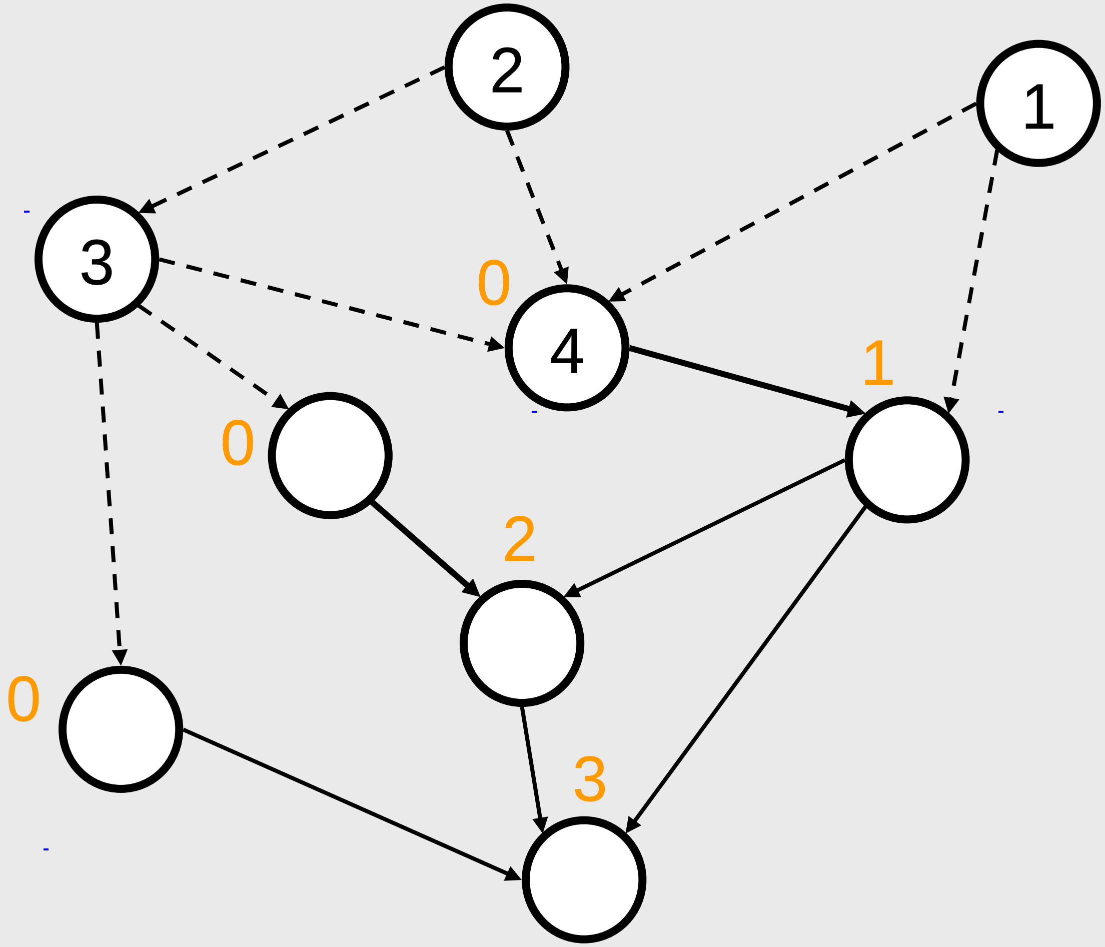
Topological Sorting Example



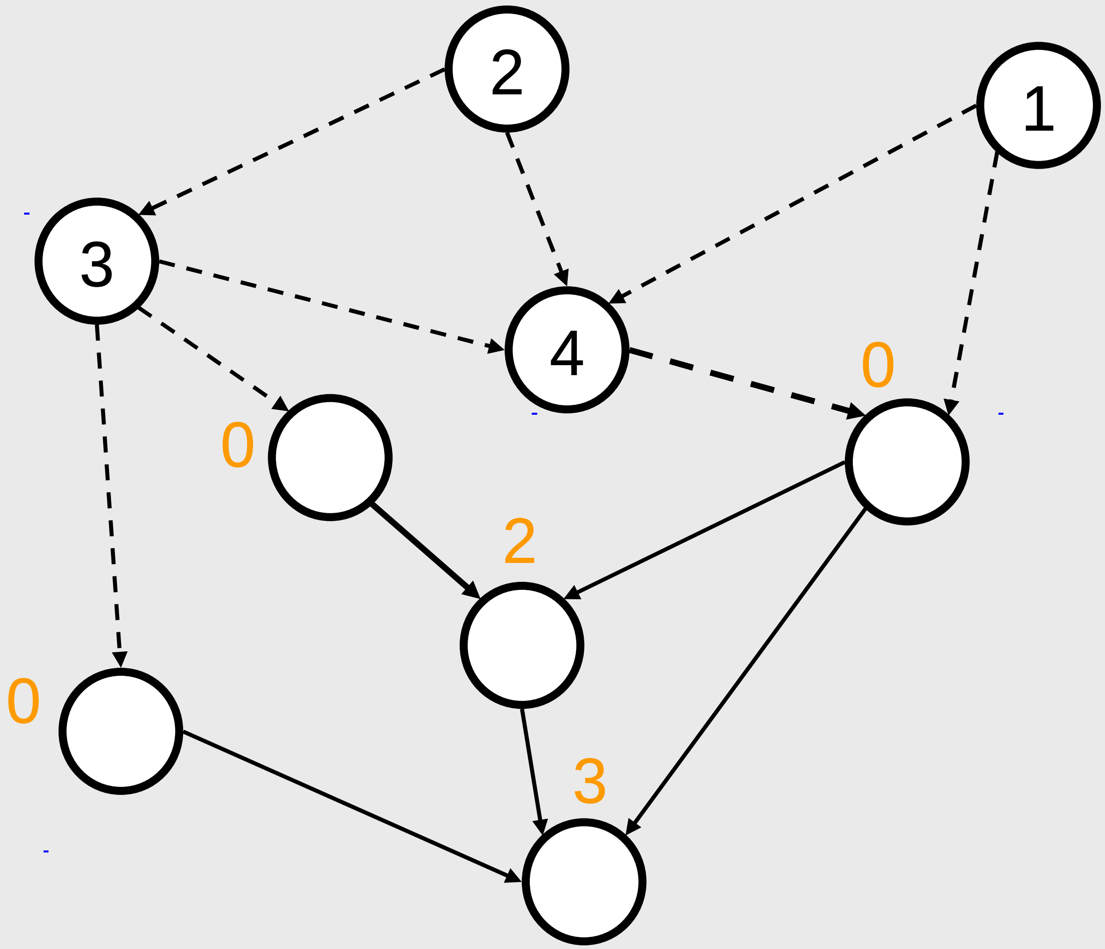
Topological Sorting Example



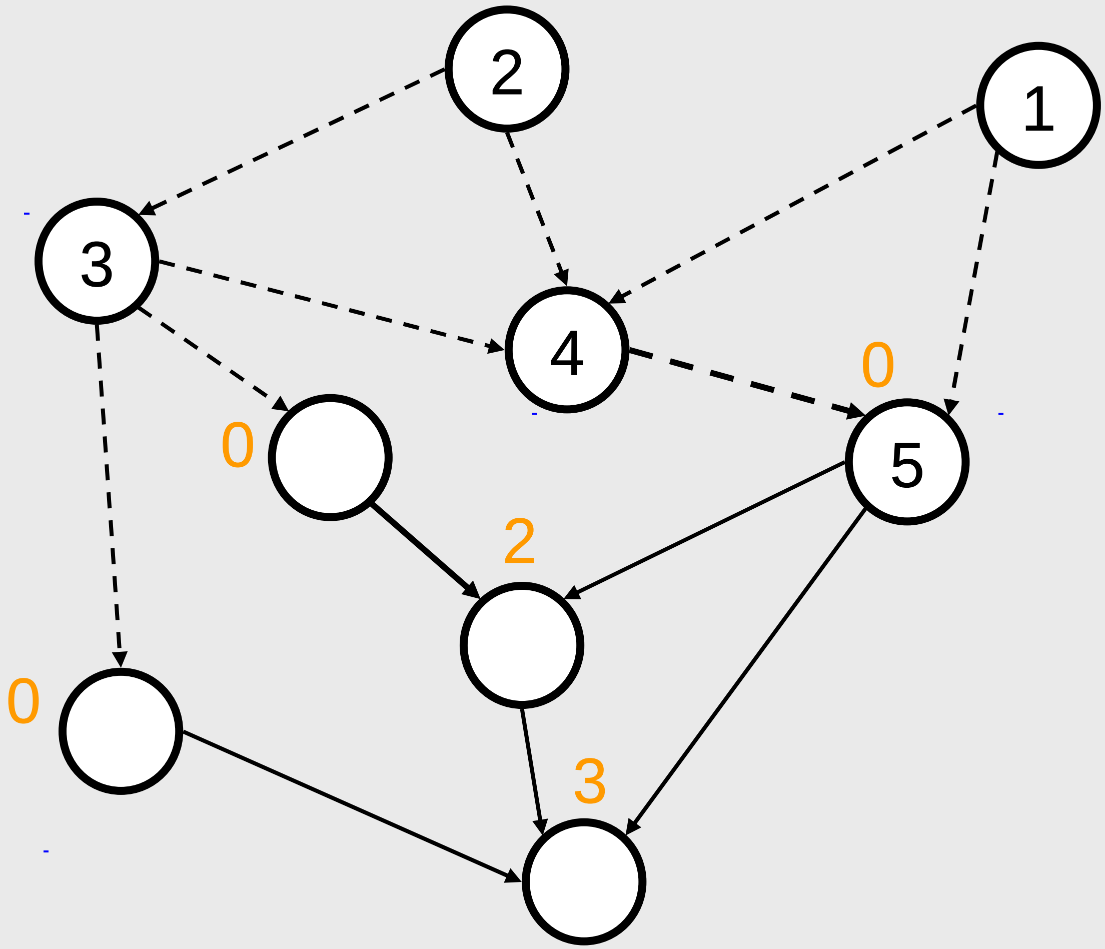
Topological Sorting Example



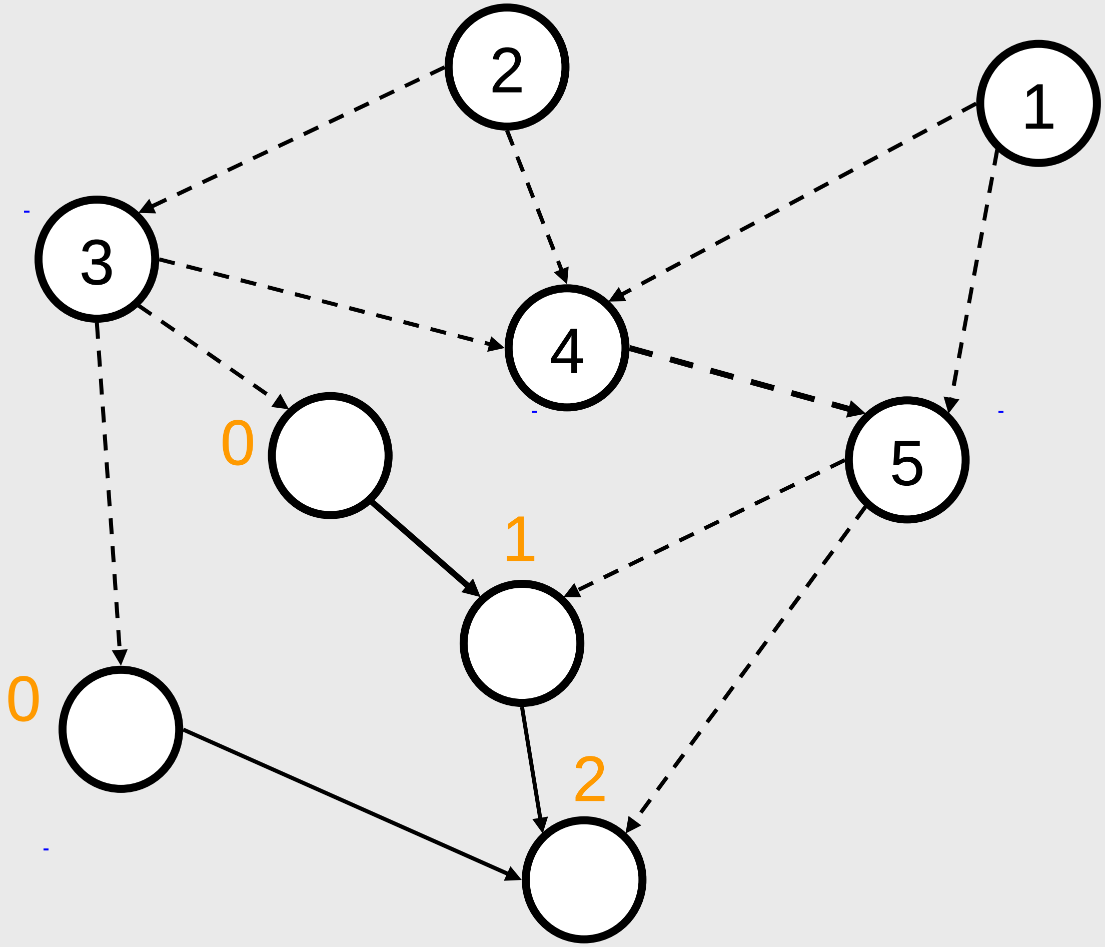
Topological Sorting Example



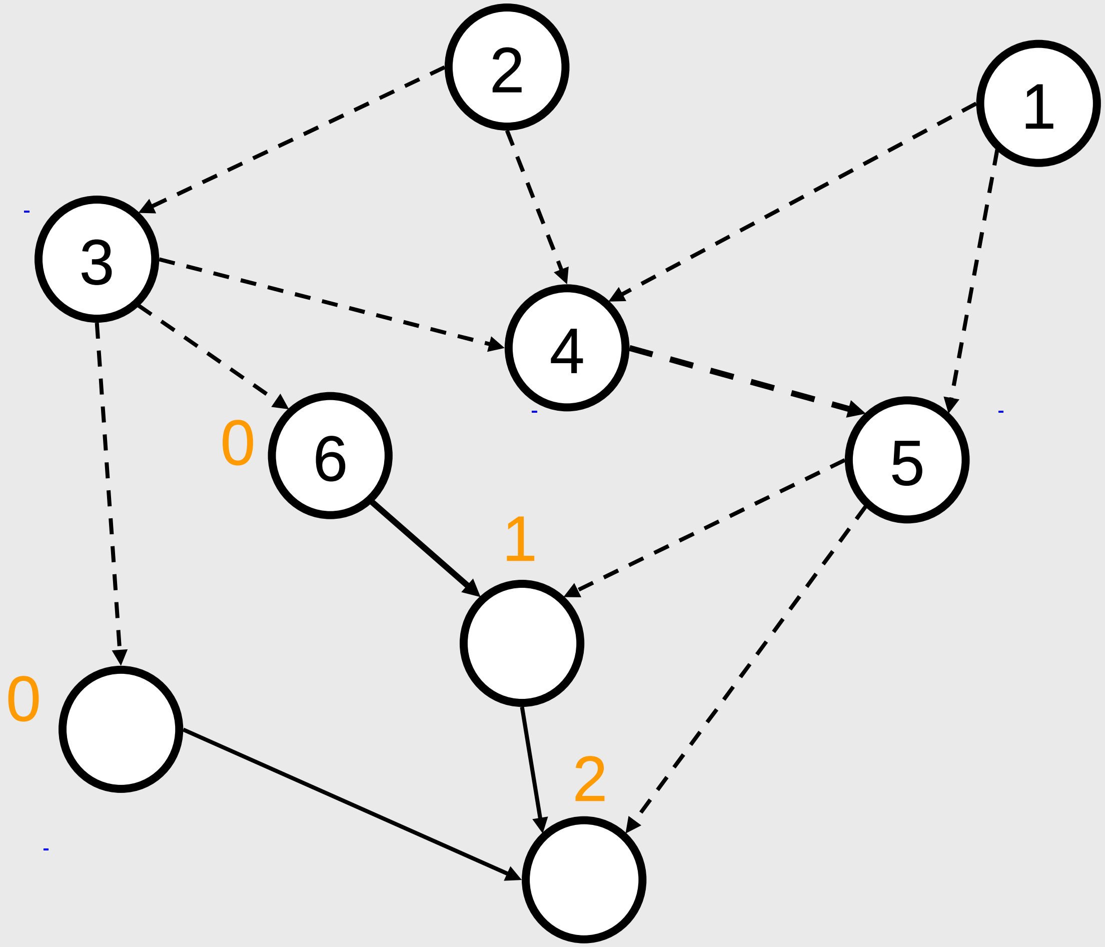
Topological Sorting Example



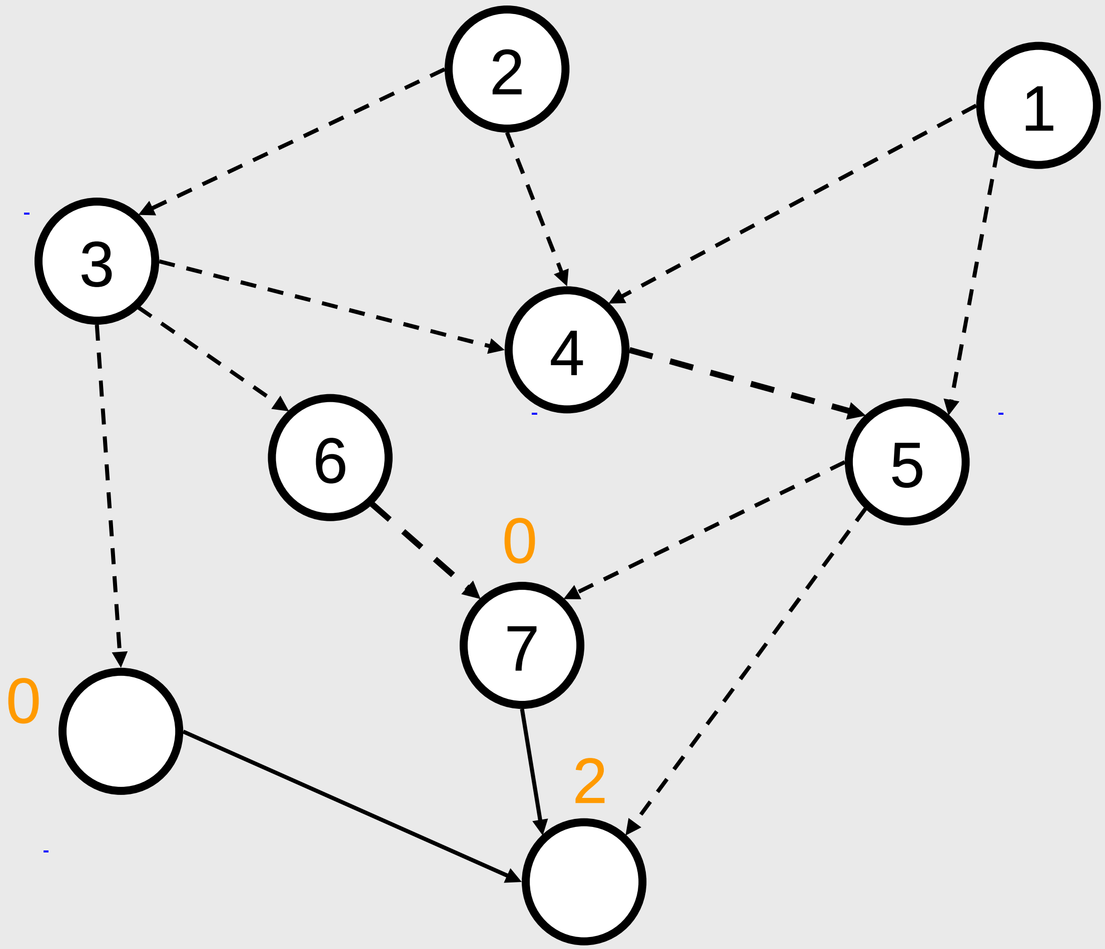
Topological Sorting Example



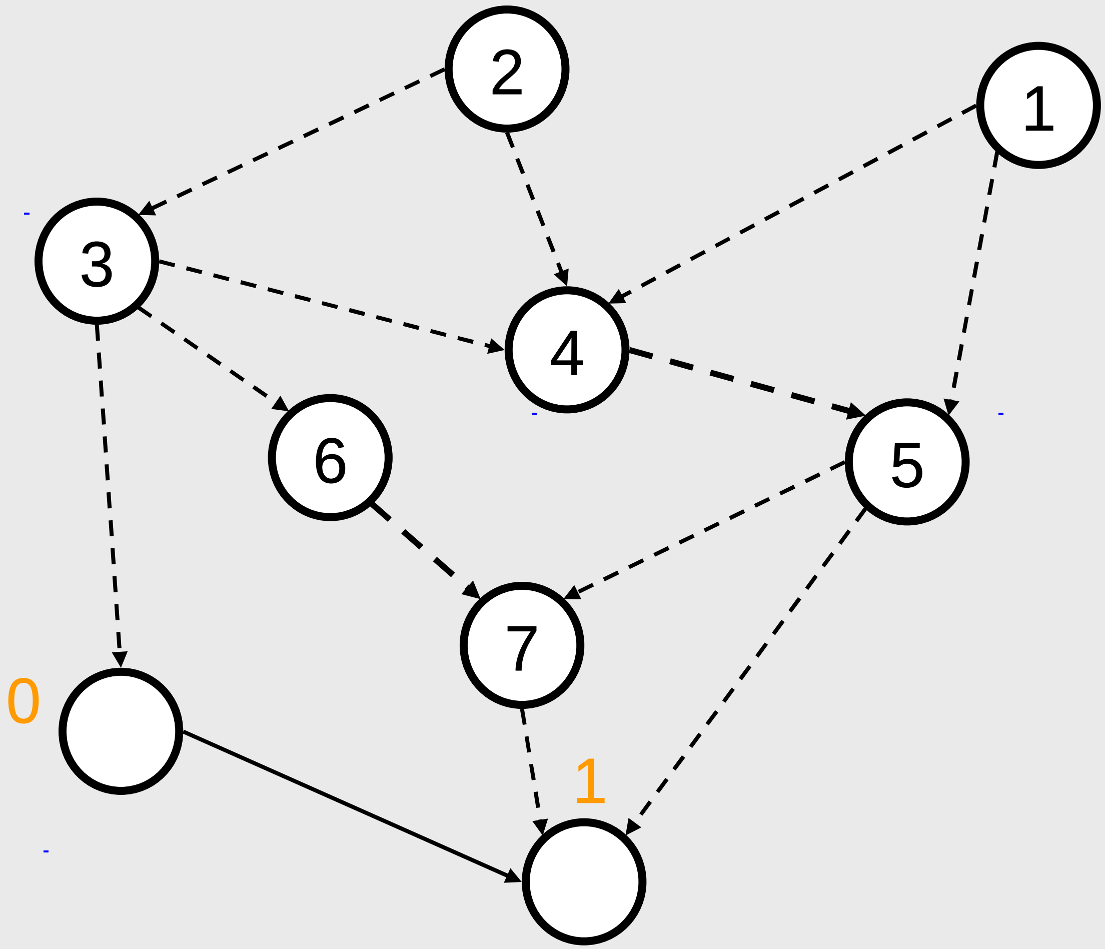
Topological Sorting Example



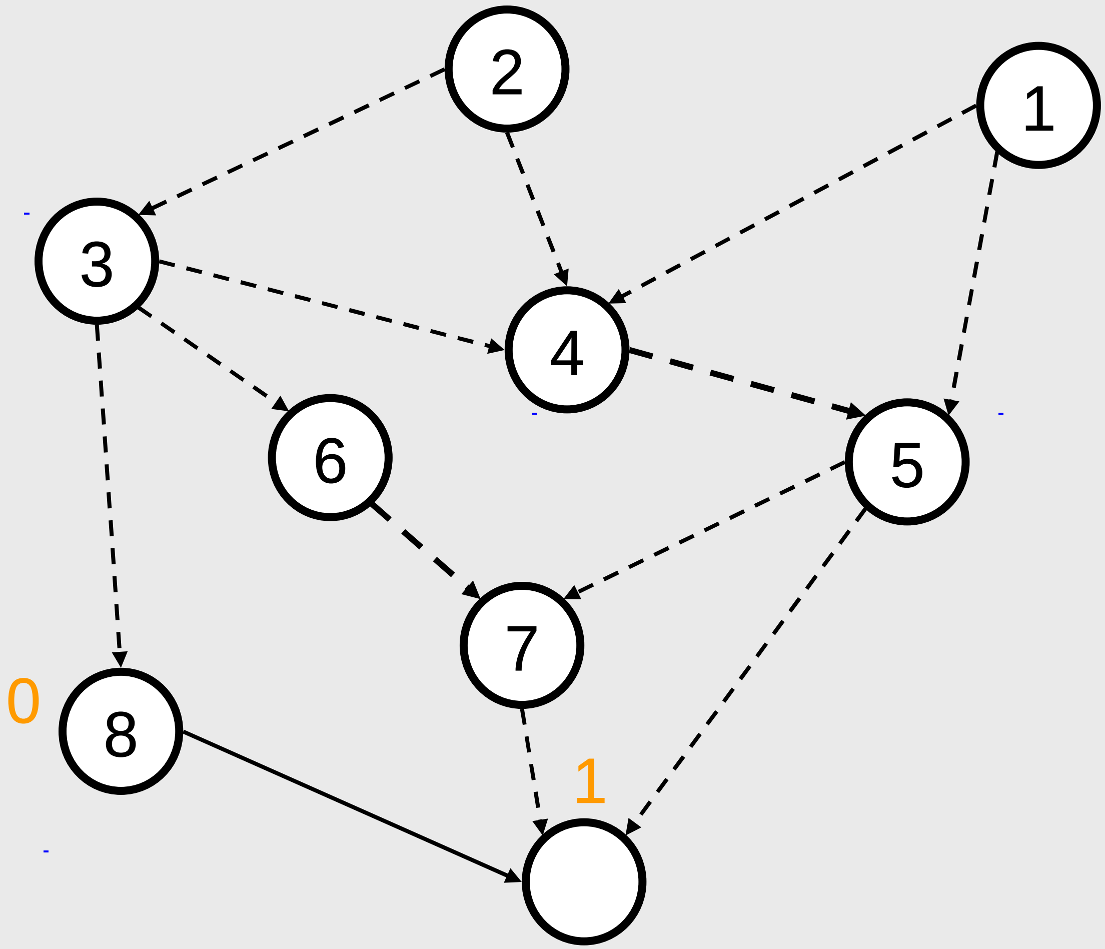
Topological Sorting Example



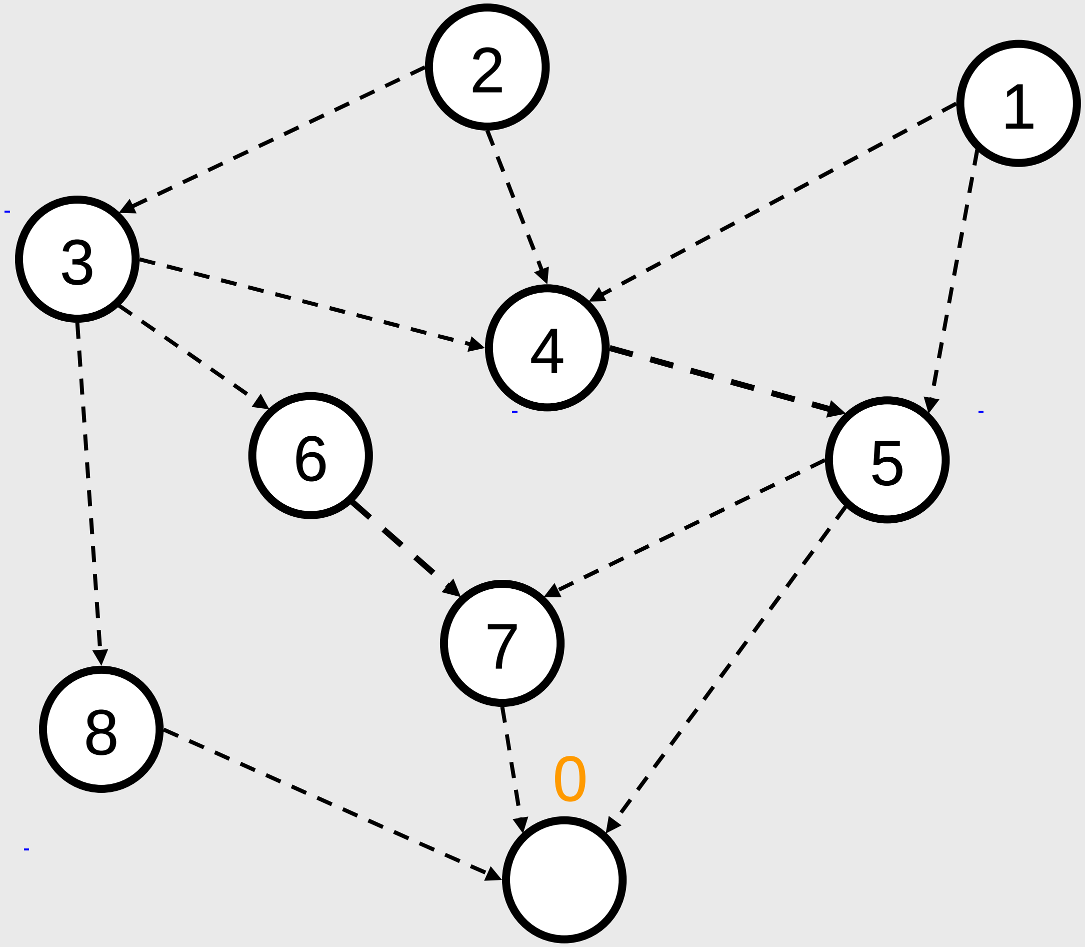
Topological Sorting Example



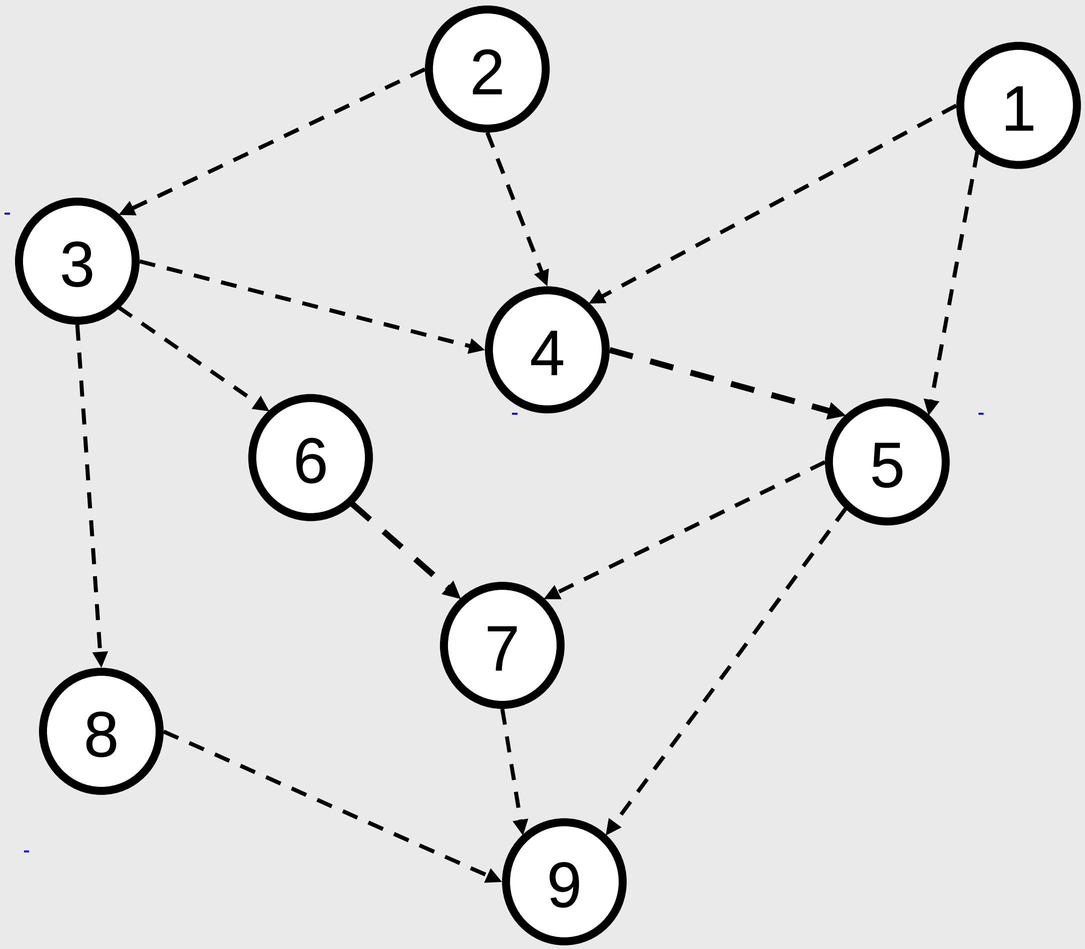
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example



Algorithm for Topological Sorting

```
Algorithm TopologicalSort(G)
    Let S be an empty stack
    for each vertex u of G do
        set its in_counter
        if in_counter = 0 then
            Push u in S
    $i \leftarrow 1$
    while S is not empty do
        Pop v from S
        Label $v \leftarrow i$
        $i \leftarrow i + 1$
        for every w adjacent to v do
            reduce the in_counter of w by 1
            if in_counter = 0 then
                Push w in S
    if (i < # of vertices)
        "Diagraph has a directed cycle"
```

Run Time: O(n+m)

Space use: O(n)